Compiler Forge

An Academic Handbook for
Code & Concepts of Compiler Design

GARODIA

Compiler Forge

Compiler Forge

An Academic Handbook for Compiler Design

Getting Started with concepts and coding for your academic excellence

by Sourav Garodia

Author and Publisher: Sourav Garodia
Printed Locally by the author
Dhaka, Bangladesh

Cover Design, Content, and Layout by: Sourav Garodia
Email: garodia15-5048@diu.edu.bd

Includes Project: Compiler Forge Toolkit
https://compiler-forge.netlify.app

Available in both print and digital formats.
First Edition: April 2025

I dedicate this book to my mother; you mean the world to me.

Brief Contents

Contents in Detail

Programming Problems Index

ACKNOWLEDGEMENTS

I would like to express my gratitude to my course teacher, Mushfiqur Rahman, for introducing me to compiler design and motivating me to write this book.

I have learned a lot from Compilers: Principles, Techniques, and Tools (2nd Edition) by Aho, Ullman, and others. Additionally, I have taken inspiration from various online resources, articles, and tutorials that helped me understand different concepts.

This book is the result of my learning, and I hope it helps others in their journey as well.

Getting Started with This Book

This book is a complete guide to Compiler Design, combining both theory and lab components into a unified format. Each chapter includes clear explanations, practical examples, exercises, and programming problems to strengthen both understanding and implementation skills.

To support hands-on learning, a GitHub repository is linked with each chapter, containing code for lab programs that can be directly run and tested. Additionally, the book introduces the Compiler Forge Toolkit, a powerful add-on for solving grammar-related problems like left recursion, left factoring, and first & follow sets.

With this integrated approach, students can smoothly learn, practice, and build compiler components step by step..

1

Introduction

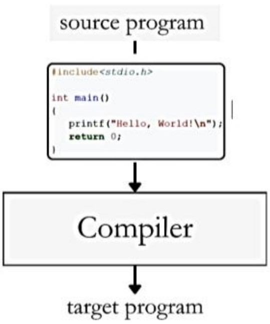
Programming languages are not just tools for writing software; they are the means by which humans communicate with machines. Every piece of software—from operating systems to web applications—relies on a programming language. However, computers do not inherently understand high-level programming languages like C, Java, or Python. Instead, they operate on machine code, a sequence of binary instructions that directly control the hardware.

Before a program can be executed, it must first be translated into machine code that the computer can understand. This translation is performed by a special program called a compiler. The role of a compiler is essential in the software development process, as it enables high-level instructions written by programmers to be transformed into optimized machine instructions that can be executed efficiently by the computer's processor.

In this chapter, we will explore the different forms of language processors, steps of language processing and understand the role of a compiler.

1.1 Compilers and Other Language Processors

A compiler is a program that reads a program written in one language—called the source language—and translates it into an equivalent program in another language—called the target language. The source language is typically a high-level programming language such as C, Java, while the target language is often machine code or assembly language; see Fig. 1.1. An essential role of the compiler is to report any errors in the source program that it detects during the translation process.



```
#include<stdio.h>

int main()
{
   printf("Hello, World!\n");
   return 0;
}
```

Figure 1.1: A compiler

If the target program is an executable machine-language program, the user can run it independently to process inputs and produce outputs. see Fig. 1.2.

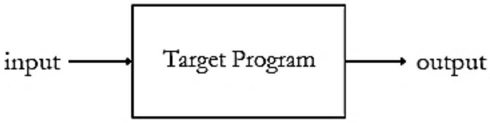


Figure 1.2: Running the target program

This approach ensures that the program executes efficiently since all translation work is completed beforehand. However, compilers have a trade-off: the compilation process can be time-consuming, as the entire program must be translated before execution.

An **interpreter** is another common type of language processor. Unlike a compiler, an interpreter does not generate a separate target program. Instead, it directly executes the operations specified in the source program, processing inputs dynamically as they are supplied by the user. See Fig 1.3.

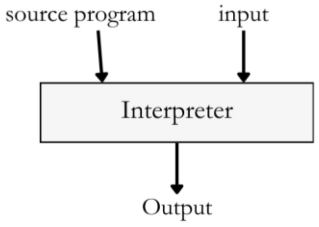


Figure 1.3: An Interpreter

Since an interpreter does not require compilation before execution, it allows for faster debugging and immediate feedback, making it ideal for scripting languages such as Python and JavaScript. However, this also means that interpreted programs generally run slower than compiled ones, as they must be translated line by line during execution.

Key Differences Between Compilers and Interpreters

While both compilers and interpreters serve the purpose of translating and executing programs, they differ in several ways:

- A compiler translates the entire program before execution, whereas an interpreter translates and executes line by line.
- Compiled programs tend to run faster because they are already converted into machine code, while interpreted programs are slower since translation happens during execution.
- Compilers detect all errors at once after scanning the entire source program, whereas interpreters stop execution as soon as they encounter an error.

1.2 The Language Processing System

The transformation of source code into an executable program is not a single-step process. Instead, it involves multiple stages, each with a specific responsibility. This series of transformations is known as the language processing system. See Fig 1.4.

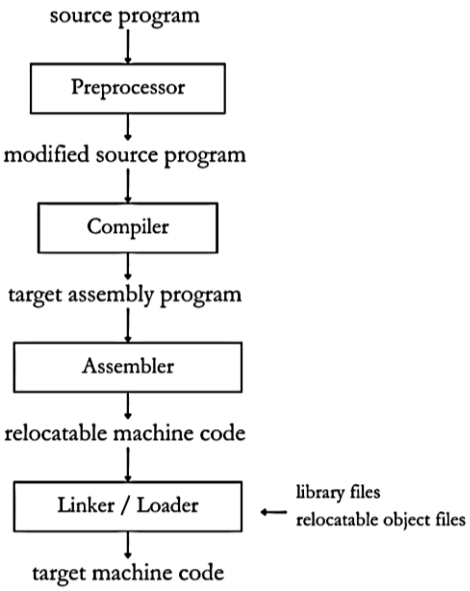


Figure 1.4: A language-processing system

The journey of a program from source code to execution typically follows these stages:

Preprocessing: Before compilation begins, a preprocessor handles tasks such as macro expansion, file inclusion, and conditional compilation. For example, in C programming, preprocessor directives like #include and #define are processed at this stage.

Compilation: The compiler takes the preprocessed source code and converts it into assembly language, a human-readable form of machine instructions. During this stage, the compiler detects syntax errors, ensuring that the program follows the grammatical rules of the language.

Assembly: The assembler translates the assembly code into machine code (binary instructions). At this point, the program is almost ready to run, but it may still rely on external libraries or functions.

Linking & Loading: Many programs depend on external libraries, such as standard I/O functions (printf, scanf) or third-party modules. The linker resolves these dependencies by combining multiple compiled files into a single executable. The loader then places the executable into memory for execution.

1.3 Role of a Compiler in Program Execution

Compilers play a fundamental role in modern computing, enabling software to run efficiently across different platforms. By translating human-readable source code into a machine-executable format, a compiler acts as a bridge between programming languages and computer hardware. Without compilers, developers would have to write code directly in machine language, making software development slow, complex, and error-prone.

It is important to note that compilation is not a single-step process. The transformation from high-level code to an executable program involves multiple stages, including analyzing the structure of the program, optimizing its performance, and generating the final executable. We will explore these stages in detail in the upcoming chapters.

In addition to traditional compilers, specialized compilers exist for just-in-time (JIT) compilation, bytecode translation, and domain-specific optimizations. For instance, Java uses the Java Virtual Machine (JVM), which compiles Java code into an intermediate bytecode before executing it on different platforms. Similarly, modern web browsers use JIT compilers to optimize JavaScript execution dynamically. These variations highlight the wider applications of compiler technology beyond traditional program translation.

As we move forward, we will explore how compilers function internally, breaking down their processes into distinct phases, each with a specific role in transforming source code into a final executable.

Lighten the bulb!

- If compilers produce faster code than interpreters, why do we still use interpreters at all?

1.4 Programming Problems

P 1.1: Write a program that takes a string as input and prints it back to the user.

The program should take a string as input and print it exactly as the user entered. Since strings may contain spaces, we need to handle multi-word input correctly.

```c
#include <stdio.h>

int main() {
    char str[100];

    printf("Enter a string: ");
    scanf("%[^\n]", str);   //Read input including spaces
    printf("You entered: %s\n", str);

    return 0;
}
```

Input	Output
Hello word!	Hello word!

For getting a better understanding of the concepts of C, see **Appendix** at **Page 137.**

P 1.2: Finding the Length of a String (Without Using Library Functions).

The strlen() function is commonly used to find the length of a string. We will loop through the string until we reach the null terminator **(\0)**, which marks the end of a string in C.

```c
#include <stdio.h>

int main() {
    char str[100];
    int length = 0;

    printf("Enter a string: ");
    scanf("%[^\n]", str);

    while (str[length] != '\0') {
        length++;
    } // Count characters until null terminator

    printf("Length of the string: %d\n", length);
    return 0;
}
```

Input	Output
Hello word!	Length of the string: 11

P 1.3: Accepting Multi-line Input and Printing It

Write a program that allows the user to enter multiple lines of text and prints them back. The program should stop reading when a semicolon (;) is encountered. We will use a special format specifier (**%[^;]s**) to read input until ; .

```c
#include <stdio.h>

int main() {
    char str[200];

    printf("Enter multiple lines (end with ';'):\n");
    scanf("%[^;]s", str);
       // Reads input until a semicolon is encountered
    printf("%s", str);

    return 0;
}
```

Input	Output
Hello word! This is Compiler Design Handbook by Sourav	Hello word! This is Compiler Design Handbook by Sourav

Exercises

Conceptual Questions

1. What is the main difference between a compiler and an interpreter?
2. Why do high-level programming languages need translators?
3. Describe the steps of the language processing system and their importance.
4. Explain why compiled programs generally run faster than interpreted programs.
5. What are some real-world applications of compiler technology outside of programming languages?

Programming Problems

1. Write a program that reverses a string manually, without using the built-in **strrev()** function.
2. Write a program that takes a string as input and prints each word of that string in new line.

Explore the codes in **GitHub**

Click or Scan the QR

2

Phases of a Compiler

A compiler is not a single-step translator that magically converts source code into machine code. Instead, it works in a sequence of well-defined phases, each responsible for a specific transformation. These phases ensure that the source code is analyzed, optimized, and translated efficiently into an executable form.

Understanding these phases is essential in compiler design, as each stage plays a crucial role in ensuring the correctness and efficiency of the final program. This chapter explores the six main phases of compilation and introduces compiler construction tools that help automate parts of this process.By the end of this chapter, you will have a clear understanding of how compilers transform human-readable programs into machine-executable code.

2.1 The Six Phases of Compilation

Compilation isn't a one-step process. Instead, a compiler performs multiple transformations on source code, ensuring it's syntactically correct, semantically valid, and optimized for execution.

These transformations occur in six major phases, categorized into two broad sections:

Analysis Phase (Front-end) : This phase understands the source code by analyzing its structure and correctness.

1. Lexical Analysis (Scanning) – Breaks the code into tokens.
2. Syntax Analysis (Parsing) – Checks if the structure follows grammar rules.
3. Semantic Analysis – Ensures meaning correctness (type checking, scope).
4. Intermediate Code Generation – Converts code into a machine-independent format.

Synthesis Phase (Back-end) : This phase generates optimized machine code for execution.

1. Code Optimization – Improves efficiency.
2. Code Generation – Produces machine-executable code.

Throughout the compilation process, two essential components work alongside the six phases: the Symbol Table and Error Handling. The symbol table is a data structure that records information about variables, functions, and other identifiers, including their types, scopes, and memory locations. It is built during Lexical and Syntax Analysis, checked in Semantic Analysis, and referenced in later stages to generate efficient code. Additionally, error handling occurs at multiple points—detecting invalid tokens in Lexical Analysis, reporting syntax violations in Parsing, and ensuring correct data usage in Semantic Analysis. These mechanisms, as shown in Figure 2.1, support the compiler in maintaining accuracy and efficiency throughout all phases.

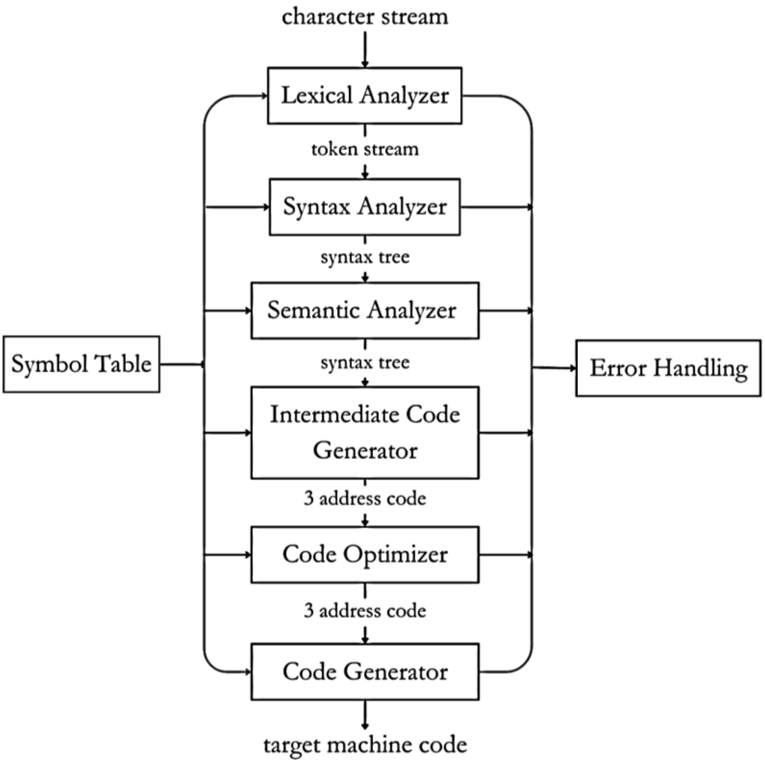


Figure 2.1: Six phases of a compiler

2.2 Lexical Analysis

The first phase, lexical analysis or scanning, converts the sequence of characters in the source code into tokens—the smallest meaningful units in a program, by deleting any white space or commentary in the source code. If a token is identified as invalid by the lexical analyzer, it produces an error.

In Lexical Analysis, the source code is broken down into lexemes, which are sequences of characters that form meaningful units in a program. Each lexeme is then classified into a token, which represents its type, such as keywords, identifiers, operators, constants, and punctuation symbols.

For example, in the statement int x = 10; , the lexemes are int, x, =, 10, and ; , which correspond to tokens like KEYWORD, IDENTIFIER, ASSIGNMENT_OPERATOR, CONSTANT, and SEMICOLON. These tokens are passed to the Syntax Analyzer, which ensures they follow the grammatical rules of the programming language.

For example, suppose a source program contains the assignment statement,

```
a = b + c * 8
```

The scanner breaks it down into tokens like this:

1. **a:** The lexeme "a" would be mapped into the token <id, 1>, and a symbol-table entry for the identifier a.
2. **=:** = is a lexeme that is mapped into the token <=>.
3. **b:** The lexeme "b" would be mapped into the token <id, 2>, and a symbol-table entry for identifier b.
4. **+:** + is a lexeme that is mapped into the token <+>.
5. **c:** The lexeme "c" would be mapped into the token <id, 3>, and a symbol-table entry for identifier c.
6. ***:** * is a lexeme that is mapped into the token <*>.
7. **8:** 8 is a lexeme that is mapped into the token <8>.

So, after lexical analysis, the statement int a = b + 5 would be represented as the following sequence of tokens:

```
<id,1> <=> <id,2> <+> <id,3> <*> <8>
```

Each token is passed to the syntax analyzer for further processing. The scanner also removes unnecessary characters, such as whitespace and comments, ensuring that only meaningful elements proceed to the next stage.

2.3 Syntax Analysis

The second phase of the compiler is syntax analysis or parsing. The parser uses the first components of the tokens produced by the lexical analyzer to create a tree-like intermediate representation and checks if they follow the rules of the programming language.

The parser builds a syntax tree, enforcing precedence rules. If a statement doesn't follow grammatical rules, the parser throws a syntax error.

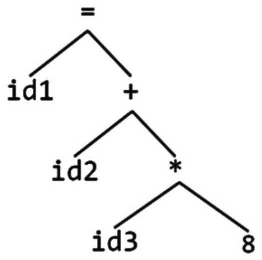


Figure 2.2: Syntax Tree or Parser Tree

2.4 Semantic Analysis

The semantic analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition. The semantic analyzer ensures type correctness, variable scope rules, and function compatibility.

An essential component of semantic analysis is type checking, which ensures that operators have matching operands. In many cases, language specifications allow certain type conversions, known as coercions. For example, a binary operator may accept either two integers or two floating-point numbers. However, if one operand is an integer and the other is a floating-point number, the compiler may automatically convert the integer to a floating-point value.

In our previous syntax tree, the integer value 8 is used with a floating-point number. The compiler will apply a type coercion to convert the integer (8) into a floating-point number before performing the multiplication operation.

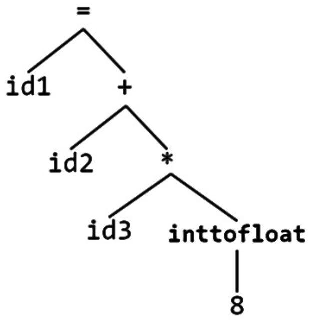


Figure 2.3: Syntax Tree in semantic analysis

This coercion is represented by the explicit conversion operation inttofloat, (see Fig. 2.3.) ensuring the operands are compatible for the arithmetic operation.

2.5 Intermediate Code Generation

Once the semantic analysis phase is over the compiler, generates intermediate code for the target machine. This intermediate representation should have two important properties: it should be easy to produce and it should be easy to translate into the target machine. An intermediate form called three-address code, which consists of a sequence of assembly-like instructions with three operands per instruction is considered generally.

Each three-address assignment instruction has at most one operator on the right side. The intermediate code for the syntax tree (Fig 2.3) having the form of 3 address code is given below:

```
tl = inttofloat(8)
t2 = id3 * tl
t3 = id2 + t2
id1 = t3
```

2.6 Code Optimization

In the code optimization phase, the intermediate code is improved to produce more efficient target code. This improvement typically focuses on increasing speed, though other goals such as reducing code size or power consumption may also be prioritized.

```
tl = id3 * 8.0
id1 = id2 + t1
```

The optimizer can simplify operations by detecting unnecessary conversions. For example, in the case of converting an integer (8) to a floating-point number, the optimizer can replace the integer with the floating-point value (8.0) directly, eliminating the inttofloat operation. This reduces redundant operations and results in more efficient code.

Additionally, if a temporary variable like t3 is used only once, the optimizer can eliminate it and directly assign the value to the target variable, resulting in a more compact and efficient intermediate code.

2.7 Code Generation

Code generation is the last and final phase of a compiler. The code generator takes as input an intermediate representation of the source program and maps it into the target language. If the target language is machine code, registers or memory locations are selected for each of the variables used by the program and instructions are translated into sequences of machine instructions that perform the same task.

For example, using registers R1 and R2, the previous intermediate code might get translated into the machine code :

```
LDF  R2, id3
MULF R2, R2, #8.0
LDF  Rl, id2
ADDF R1, R1, R2
STF  id1, R1
```

2.8 Symbol Table

Throughout the phases of the compiler, a symbol table is maintained. The symbol table is a data structure containing a record for each variable name, with fields for the attributes of the name. The data structure should be designed to allow the compiler to find the record for each name quickly and to store or retrieve data from that record quickly.

2.9 Compiler-Construction Tools

Specialized tools are designed for specific compiler phases. These tools use specialized languages and algorithms to simplify development. Some commonly used tools include:

1. **Parser generators:** Automatically create syntax analyzers from a language's grammar.
2. **Scanner generators**: Generate lexical analyzers from regular expressions.
3. **Syntax-directed translation engines:** Produce routines for generating intermediate code from parse trees.
4. **Code-generator generators:** Create code generators to translate intermediate code to machine language.
5. **Data-flow analysis engines:** Help gather information about how data flows in a program, aiding in code optimization.
6. **Compiler-construction toolkits:** Provide integrated routines for building various compiler phases.

2.10 Errors in Compiler

Throughout the various phases of a compiler, errors can be detected and categorized into different types. These errors can occur at different stages, such as lexical analysis, syntax analysis, semantic analysis, and during optimization. Below are the common types of errors found in the compiler phases:

Types of Errors

Lexical Errors: Occurs when there is a typo in the code, such as a misspelled identifier or incorrect token. *Example*: Typing *innt* instead of *int*.

Syntactical Errors: Occurs when the code structure does not follow the syntax rules of the language, such as missing semicolons or unbalanced parentheses. *Example*: Missing closing parentheses in the main function declaration.

Semantical Errors: Happens when the program's logic is inconsistent with the rules of the language, such as assigning incompatible values. *Example*: Assigning a value to an array index that exceeds its declared size or performing incompatible operations.

Logical Errors: These are typically issues in the program's logic that cause incorrect behavior, such as infinite loops or unreachable code. *Example*: A loop that runs indefinitely or a block of code that can never be executed.

Lighten the bulb!

- Why is intermediate code generation necessary, instead of translating directly from source code to machine code?

Why doesn’t my code run!

A segment of code having multiple error and their types are explained below for better understanding.

```
#include<stdio.h>
int main{
  innt a[2]={2,4,6}, b=1;
  sum = a[b]+b
  prntf("Result is: %f, sum);

  return b;
}
```

Here,

- **main = Syntactical error**: Missing parentheses after main
- **innt = Lexical error**: Incorrect spelling of int.
- **a[2] = Semantic error**: Incorrect array size (declared 2 but initialized with 3 elements.
- **sum = Semantic error**: sum is not declared before use.
- **; = Syntactical error**: Missing semicolon after sum = a[b] + b.
- **prntf() = Lexical error**: Incorrect spelling of printf().
- **%f = Semantic error**: %f expects a float or double, but sum is not declared.
- **“ = Syntactical error**: Missing closing quote in the printf statement.

In later chapters, we will discuss specific types of errors in greater detail and the methods used for error handling in compilers.

2.11 Programming Problems

P 2.1: Write a program that counts the whitespace in a string.

The program should take a string as input and count whitespace like ‘ ’, ‘\t’, ‘\n’.

```c
#include <stdio.h>

int main() {
    char sentence[100];
    int count = 0;

    printf("Input sentence: ");
    scanf("%99[^\n]", sentence);

    for (int i = 0; sentence[i] != '\0'; i++) {
        if (sentence[i] == ' ' || sentence[i] == '\t' ||
sentence[i] == '\n') {
            count++;
        }
    }
    printf("Number of whitespaces: %d\n", count);

    return 0;
}
```

Input	Output
This is Compiler Design Handbook by Sourav	Number of whitespaces: 5

P 2.2: Write a program that eliminates extra whitespace in a string.

From the string eliminate whitespace like ‘ ’ , ‘\t’ , ‘\n’ but only if it is extra, that is we have to check if they are more than one or not.

```c
#include <stdio.h>
void eliminateExtraWhitespace(char *str)
{
    int i = 0, j = 0;

    while (str[i] != '\0' && str[i] == ' ')
    {
        i++;
    }

    while (str[i] != '\0')
    {
        if (str[i] != ' ' || (i > 0 && str[i - 1] != ' '))
        {
            str[j++] = str[i];
        }
        i++;
    }
    str[j] = '\0';
}

int main()
{
    char str[100];
    int Count = 0;

    printf("Input sentence: ");
    scanf("%99[^\n]", str);
    eliminateExtraWhitespace(str);

    printf("Modified sentence: '%s'\n", str);

    return 0;
}
```

Input	Output
This is Compiler Design Handbook	This is Compiler Design Handbook

P 2.3: Write a program that eliminates whitespace in a string.

The program should take a string as input and eliminate whitespace like ‘ ’ , ‘\t’ , ‘\n’.

```
#include <stdio.h>
void removeWhitespace(char *str)
{
    int i = 0, j = 0;
    while (str[i] != '\0')
    {
     if (str[i] != ' ' && str[i] != '\t' && str[i] != '\n')
      {
        str[j++] = str[i];
      }
      i++;
    }
    str[j] = '\0';
}
int main()
{
    char str[100];

    printf("Enter a sentence: ");
    scanf("%99[^\n]", str);
    removeWhitespace(str);
    printf("Sentence without whitespace: '%s'\n", str);
    return 0;
}
```

Input	Output
This is Compiler Design Handbook	Sentence without whitespace: ThisisCompilerDesignHandbook

Exercises

Conceptual Questions

1. Demonstrate the phases of a compiler.
2. Apply the six phases of a compiler for this **a = b * c + 8** source program.
3. Explain the errors of compiler.
4. Identify the errors from the codes given in GitHub section Q2.4

Programming Problems

1. Write a program that eliminates Eliminate special character in a string.
2. Write a program that counts whitespaces, digits, punctuation in a string.
3. Write a program that counts the number of spaces, tabs, and newlines in a given input string.

Explore the codes in **GitHub**

Click or Scan the QR

3

Lexical Analysis & Context-Free Grammar

When a compiler processes source code, the first thing it needs to do is break it down into smaller components that can be analyzed. This is where Lexical Analysis comes into play.

The Lexical Analyzer reads the program as a stream of characters and groups them into lexemes—meaningful sequences of characters—before categorizing them into tokens, which are standardized representations of code elements. However, while the Lexical Analyzer can identify what each part of the code is (such as a variable name or an operator), it cannot determine whether the code is structured correctly. This limitation requires the introduction of Syntax Analysis, which is based on Context-Free Grammar (CFG).

In this chapter, we will take a deep dive into Lexical Analysis, understand how tokens, lexemes, and patterns work, explore types of lexical tokens, and examine the role of the Lexical Analyzer in handling errors. We will then introduce CFG, which forms the basis for syntax checking and parsing. Finally, we will study derivations, parse trees, and ambiguity detection, along with a brief discussion on transition diagrams.

3.1 Lexical Analysis: The First Step of Compilation

Before a program can be understood by a computer, it must be broken down into recognizable pieces. This is what Lexical Analysis does. It scans the source code and groups sequences of characters into tokens that the compiler can process.

Lexemes, Tokens, and Patterns

A lexeme is a sequence of characters that form a valid unit in a programming language. Each lexeme is classified into a token, which represents its category. Tokens allow the compiler to understand the function of different parts of the code.

The sequence of characters matched by a pattern to form the corresponding token or a sequence of input characters that comprises a single token is called a lexeme. eg- "float", "abs_zero_Kelvin", "=", "-", "273", ";" .

```
int a = b + 8;
```

```
int KEYWORD
a IDENTIFIER
= ASSIGNMENT OPERATOR
b IDENTIFIER
+ ARITHMETIC OPERATOR
8 CONSTANT
; PUNCTUATION
```

To identify these tokens, the Lexical Analyzer follows patterns that define how a lexeme should be structured. By matching lexemes to patterns, the

Lexical Analyzer efficiently classifies elements of a program, making it easier for the compiler to process them.

Types of Lexical Tokens

Every programming language consists of several types of tokens, each serving a specific role. The five major categories of tokens are:

1. **Keywords** – Reserved words that have special meanings, such as int, float, return, if, else, and while
2. **Identifiers** – Names assigned to variables, functions, arrays, and other user-defined elements, such as sum, count, or calculateArea
3. **Operators** – Symbols used for mathematical, relational, or logical operations, such as +, -, *, /, &&, and ==.
4. **Constants** – Fixed values in a program, including numbers (10, 3.14) and characters ('A').
5. **Punctuation & Special Symbols** – Characters such as ;, ,, {}, and () that define the structure of a program.

Each of these tokens has strictly defined rules for formation and usage, enforced by the Lexical Analyzer. There are some none tokens also like : Comments, preprocessor directive, macros, blanks, tabs, newline etc. For example:

Whitespace – space(' ') tab('\t') newline('\n')

Comments – /*this is not a token*/

3.2 The Role of the Lexical Analyzer

The Lexical Analyzer does much more than simply split the source code into tokens. It also plays a crucial role in preprocessing the input to make it easier for later stages of compilation. Some of its key functions include:

- Filtering out whitespace and comments, which are ignored by the compiler.
- **Handling input buffering** to improve efficiency when scanning long programs.
- **Storing token information** in a **symbol table**, which is used later by the parser.
- **Detecting and handling lexical errors**, ensuring only valid tokens proceed to the next phase.

3.3 Lexical Errors and Recovery Methods

Despite the structured nature of programming languages, errors are inevitable when writing code. The Lexical Analyzer must be able to detect and recover from lexical errors without stopping the entire compilation process.

Some common types of **lexical errors** include:

- **Illegal Characters** – Using symbols that are not part of the language, such as @ in int x@ = 10;
- **Misspelled Keywords** – Writing flot instead of float.
- **Unclosed Strings** – Forgetting to close a string, like "Hello.
- **Invalid Identifiers** – Using 3num as a variable name, which starts with a number.

Lexical Error Recovery Strategies

To recover from lexical errors, a compiler may use one of the following strategies:

- **Panic Mode Recovery** – Skip input until the next recognizable token (e.g., ignore @ and continue).
- **Deletion** – Remove the offending character and continue parsing (e.g., delete @ in x@ = 10;).
- **Insertion** – Insert a missing token to correct syntax (e.g., add " at the end of an unclosed string).
- **Replacement** – Replace an incorrect token with a valid one (e.g., change flot to float).
- **Transposition** – Swap adjacent characters if they seem to be a typographical error (e.g., foat → float).

These error-handling techniques ensure that minor mistakes do not disrupt the entire compilation process.

Limitations of Lexical Analysis

While the Lexical Analyzer can break a program into **meaningful tokens**, it **cannot check whether those tokens are arranged correctly**. This limitation arises because **regular expressions are not powerful enough** to enforce complex syntax rules. This is where Syntax Analysis comes into play, and it is based on Context-Free Grammar (CFG).

3.4 Context-Free Grammar (CFG)

A programming language is not just a collection of tokens—it has rules that determine how tokens can be combined to form valid statements. These rules are defined using Context-Free Grammar (CFG), which provides a formal way to describe the syntax of a language.

In simple terms, a CFG is a set of rules that tells the compiler what sequences of tokens are valid and how they should be structured. While Lexical Analysis identifies individual words (tokens), CFG ensures that those words form meaningful sentences according to the rules of the language.

For example, in C, the following statement is valid:

```
int x = 10;
```

However, the following statement is invalid:

```
int = 10 x;
```

Even though both statements contain valid tokens, the second statement does not follow the grammar of the language, and thus it results in a syntax error. The compiler's Syntax Analyzer uses CFG to check whether token sequences obey the language rules.

3.6 Components of a Context-Free Grammar

A CFG consists of four key components that define how a language is structured.

$$\mathbf{G = (V, \Sigma, P, S).}$$

A set of non-terminals (V). Non-terminals are syntactic variables that denote sets of strings. The non-terminals define sets of strings that help define the language generated by the grammar.

A set of tokens, known as terminal symbols (Σ). Terminals are the basic symbols from which strings are formed.

A set of productions (P). The productions of a grammar specify the manner in which the terminals and non-terminals can be combined to form strings. Each production consists of a non-terminal called the left side of the production, an arrow, and a sequence of tokens and/or on- terminals, called the right side of the production.

One of the non-terminals is designated as the start symbol (S); from where the production begins.

For example : G = (V, Σ, P, S)

Where:

- V = { Q, Z, N }
- Σ = { 0, 1 }
- P = { Q → Z | Q → N | Q → Ɛ | Z → 0Q0 |N → 1Q1 }
- S = { Q }

3.7 Derivations and Parse Trees

A derivation is the step-by-step expansion of a non-terminal into its corresponding terminals, following the CFG rules. There are two ways to perform derivations:

Leftmost Derivation (LMD): Expands the leftmost non-terminal first. If the sentential form of an input is scanned and replaced from left to right, it is called left-most derivation.

Rightmost Derivation (RMD): Expands the rightmost non-terminal first. If we scan and replace the input with production rules, from right to left, it is known as right-most derivation.

Derivation of **id + id * id** where, Production rule :

E → E + E
E → E * E
E → id

LMD	RMD
E → E * E	E → E + E
E → E + E * E	E → E + E * E
E → id + E * E	E → E + E * id
E → id + id * E	E → E + id * id
E → id + id * id	E → id + id * id

Both derivations lead to the **same result**, but their order of expansion differs. The **parse tree** represents this visually, ensuring that operators are applied in the correct sequence.

Parse Tree

A parse tree is a graphical depiction of a derivation. It is convenient to see how strings are derived from the start symbol. A parse of the above LMD is given aside in fig 3.1.

Start symbol of the derivation becomes the root of the parse tree. In a parse tree all leaf nodes are terminal and all internal nodes are non-terminals. In order traversal of the parse tree gives the original input string.

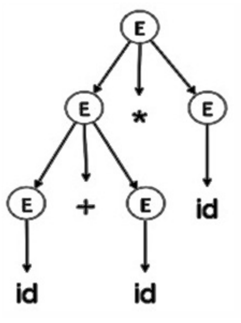


Figure 3.1 Parse Tree

3.8 Ambiguity in Grammar

A grammar is ambiguous if it allows multiple parse trees (left or right derivation) for the same expression. This can lead to different interpretations of the same statement, which is undesirable in programming languages.

Example : $E \rightarrow E+E$
$E \rightarrow E–E$
$E \rightarrow id$

For the string **id + id– id**, the above grammar generates two parse trees, and denotes the this is ambiguous.

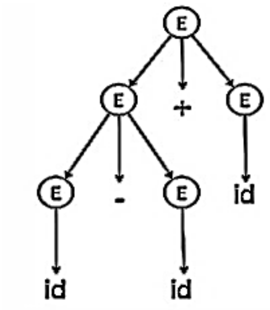


Figure 3.2 LMD

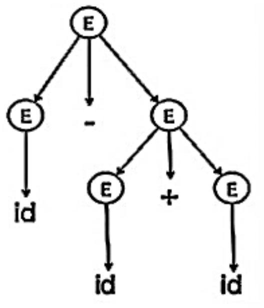


Figure 3.3 RMD

Another example for better understanding, Build the parse tree for the arithmetic expression "**a+a+a**" using the expression grammar:

E → E+E | a

LMD	**RMD**
E → E * E	E → E + E
E → E + E * E	E → E + E * E
E → a + E * E	E → E + E * a
E → a + a * E	E → E + a * a
E → a + a * a	E → a + a * a

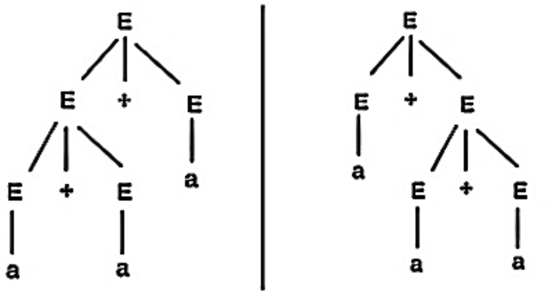


Figure 3.4 LMD Figure 3.5 RMD

This is ambiguous since the trees are not identical.

Nor for understanding unambiguous, Build the parse tree for the arithmetic expression "**()()**" using the expression grammar:

S → S S | (S) | ε

LMD	**RMD**
E → SS	E → SS
E → (S) S	E → S (S)
E → ( ) S	E → S ( )
E → ( ) (S)	E → (S) ( )
E → ( ) ( )	E → ( ) ( )

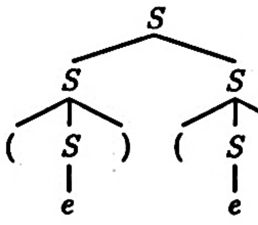


Figure 3.6 LMD

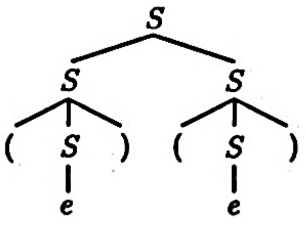


Figure 3.7 RMD

This is unambiguous or not ambiguous since the trees are identical in both LMD and RMD.

3.9 Transition Diagrams from CFG

A transition diagram is a visual representation of how a CFG can generate valid sentences. It consists of states representing different parts of an expression and transitions indicating how symbols change state.

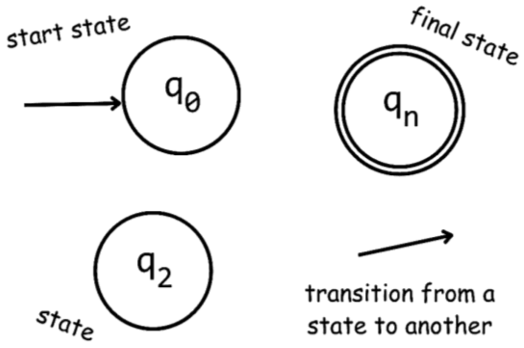


Figure 3.8 Basic signs of transition diagram

For example, a transition diagram for arithmetic expressions might look like this see figure 3.9.

Where the production rule is as follow:

E → +TE`|D

E` → E + T

D → 1 | 2 | + | -

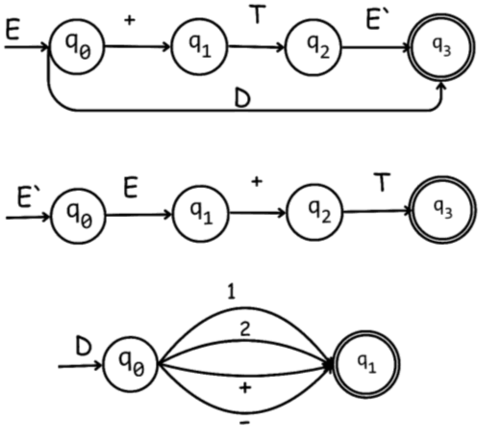


Figure 3.9 Transition diagram

In the next chapter, we will learn more about the complex transition diagrams through regular expressions and conditional statement or languages.

Lighten the bulb!

- Why do some languages allow else if but not elif, while others use elif instead of else if?

3.10 Programming Problems

P 3.1: Write a program that prints the initial.

Read the string and print the letter next to whitespace.

```c
#include <stdio.h>

int main()
{
    char a[100];
    int i = 0, j = 0;

    gets(a);
    printf("%c", a[0]);

    for (i = 1; a[i]; i++)
    {
        if (a[i] == ' ')
        {
            printf("%c", a[i + 1]);
        }
    }

    printf("\n");

    return 0;
}
```

Input	Output
United States of America	USoA

P 3.2: Write a program that finds and prints multiple line comment.

Read the string and look for /* where the multi-line comment starts and then */ to stop.

```c
#include <stdio.h>
#include <string.h>
int main()
{
    char c, a[100], b[100];
    int j = 0, len, k;

    scanf(" %[^~]", a);
    len = strlen(a);

    for (k = 0; k <= len - 1; k++)
    {
        if (a[k] == '/' && a[k + 1] == '*')
        {
            k += 2;
            while (a[k] != '*' && a[k + 1] != '/')
            {
              b[j++] = a[k++];
            }
        }
    }
    b[j] = '\0';
    printf("%s\n", b);
    return 0;
}
```

Input	Output
Hello /* this is multi-line comment */	this is multi-line comment

Exercises

Conceptual Questions

1. What is the main difference between a compiler and an interpreter?
2. Why do high-level programming languages need translators?
3. Describe the steps of the language processing system and their importance.
4. Explain why compiled programs generally run faster than interpreted programs.
5. What are some real-world applications of compiler technology outside of programming languages?

Programming Problems

1. Write a program that finds and prints single line comment.
2. Write a C program that will take single/multiple lines comments in an input String and count & print the number of comments as output.
3. Write a program that will eliminate single line comment from a given input.
4. Write a C program that will count the number of line/s in a given input.

Explore the codes in **GitHub**
Click or Scan the QR

4

Regular Expressions

In the previous chapter, we discussed Lexical Analysis and how it breaks down the source code into meaningful tokens. However, the Lexical Analyzer must have a systematic way of recognizing and classifying tokens efficiently. This is where Regular Expressions (REs) come in.

Regular expressions provide a concise and precise way to define patterns for recognizing keywords, identifiers, numbers, and other syntactic structures. In this chapter, we will explore how regular expressions are used to define tokens in a programming language. In this chapter, we will understand what regular expressions are and how they define token patterns. Learn different operators in REs like *, +, and |. See real-world examples of language-to-RE conversion and Implement programs to recognize tokens using regular expressions.

4.1 Understanding Regular Expressions

A Regular Expression (RE) is a symbolic representation of a pattern used to match sequences of characters in text. It is particularly useful in defining the structure of tokens in a programming language.

Operators of Regular Expressions

Regular expressions use special symbols to define complex patterns. Here are some fundamental operators:

1. **Literals**: Directly match specific characters. Example: The RE a denotes the character a.
2. **Concatenation**: Joins multiple expressions in sequence. Example: ab denotes "ab", "abc", but not "acb".
3. **Alternation** (|): Represents a choice between alternatives. Example: a | b denotes either "a" or "b".
4. **Kleene Star** (*): Represents zero or more repetitions of the preceding element. Example: a* denotes "", "a", "aa", "aaa", etc.
5. **Kleene Plus** (+): Represents one or more repetitions. Example: a+ denotes "a", "aa", "aaa", but not "".
6. **Grouping** (()): Used to group subexpressions. Example: (ab)+ denotes "ab", "abab", "ababab", etc.

In regular expression zero or empty is indicated through "ε"

4.2 Regular Expressions for Token Recognition

Using regular expressions, we can define common tokens found in programming languages:

Identifiers and Keywords

A valid identifier in most languages starts with a letter and can contain letters, digits, and underscores. The corresponding RE is:

```
[A-Za-z_][A-Za-z0-9_]*
```

Integer and Floating-Point Numbers

The Regular expression to recognize integer constants, we use:

```
[0-9]+
```

The Regular expression to recognize floating-point numbers with a decimal point:

```
[0-9]+\.[0-9]+
```

Operators and Punctuation

The RE to recognize arithmetic operators (+, -, *, /):

```
\+|\-|\*|\/
```

The RE to recognize assignment operators (=, +=, -=, *=):

```
=|\+=|\-=|\*=
```

To recognize punctuation symbols (;, ,, {}, [], ()):

```
[;,\{\}\[\]\(\)]
```

These REs help the Lexical Analyzer classify symbols correctly.

4.3 Language-to-Regular Expression Conversion

From a statement regular expressions can be formed. Some REs is given below for better understanding of this concept.

1. Set of all strings consisting of four letters

   ```
   LLLL  →  [A-Za-z][A-Za-z][A-Za-z][A-Za-z]
   ```

2. Set of all strings beginning with a letter, followed by any number of letters or digits

   ```
   L(L U D)*
   ```

3. Set of all strings having at least one "ab"

```
(ab)+
```

4. Set of all strings containing an even number of "aa"

```
(aa)*
```

5. Set of all strings containing an odd number of "bb"

```
b(bb)*
```

6. Set of all strings having zero or more instances of a or b starting with aa

```
(aa)(a | b)*
```

7. Set of all strings having even number of aa and even number of bb

```
(a |b)* (bb)
```

8. Set of all strings having zero or more instances of a or b starting with aa and ending with bb

```
(aa) (a | b)* (bb)
```

9. Set of all strings of a’s and b’s that do not contain the substring abb.

```
b* (a (ε|b))*
```

10. Set of all strings of a’s and b’s that contain at most two b's.

```
a* (ε|b) a* (ε|b) a*
```

11. Set of all strings of a’s and b’s that do not contain the subsequence abb.

```
b* a*(ε|b) a*
```

4.4 Regular Expression to Language Conversion

Now considering **L = {a, b}** some regular expression that can be made are:

- `a  | b` Denotes the set of {a, b} having a or b
- `(a|b)(a|b)` Denotes {aa, ab, ba, bb}, the set of all strings of a's and b's of length two
- `a*` Denotes the set of all strings of zero or more a's , i. e., {ɛ, a, aa, aaa, …..}
- `(a|b)* or (a*|b*)*` Denotes the set of all strings containing zero or more instances of an a or b, that is, the set of all strings of a's and b's.
- `a|a*b` Denotes the set containing the string a and all strings consisting of zero or more a's followed by a b.

4.5 Transition Diagram

While regular expressions provide a symbolic way to define token patterns, they do not visually represent the step-by-step recognition process. This is where Transition Diagrams come into play.

A Transition Diagram is a graphical representation of how an input string is processed to determine whether it belongs to a specific language (i.e., whether it matches a given regular expression).

Basic Symbols of a Transition Diagram

A transition diagram consists of the following main elements:

1. **States** (Nodes**):** Represented as circles. Each state indicates a position in the recognition process.
2. **Transitions** (Arrows): Directed edges between states, showing how the automaton moves based on input characters.
3. **Start State**: The initial state, marked with an incoming arrow from nowhere.

4. **Final State**: A double circle, indicating that a valid token has been recognized.
5. **Edges with Labels**: Labels on transitions represent valid input characters that allow the automaton to move to the next state.

For reference see fig 3.8. in the previous chapter.

Basic Transition Diagrams from Regular Expression

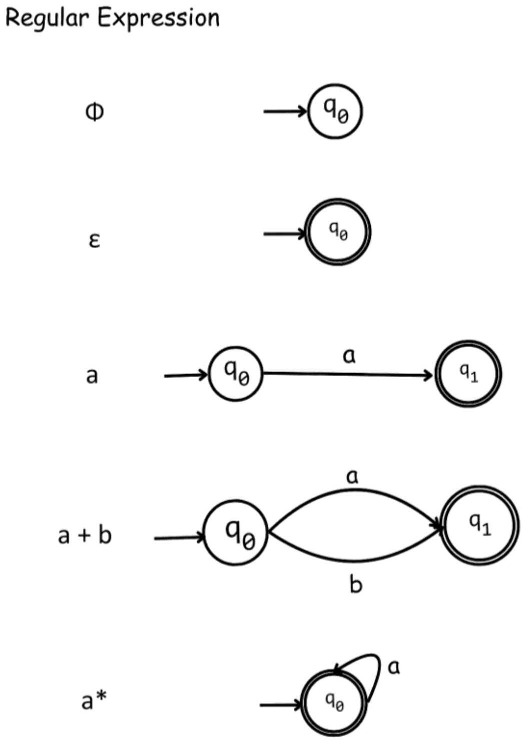


Figure 4.1 Basic Regex to Transition Diagram

In this chapter, we explored Regular Expressions (REs) and their role in Lexical Analysis, helping us define and recognize different types of tokens such as keywords, identifiers, and operators. We learned how various RE constructs—including alternation, concatenation, and repetition—allow us to specify token patterns concisely. Additionally, we discussed how REs can be converted into transition diagrams, which provide a visual representation of token recognition.

To deepen your understanding, consider exploring more examples of REs and practicing with online regex testers. Watching tutorials on YouTube or using interactive tools can also help reinforce these concepts. Mastering regular expressions is an essential step toward building a strong foundation in compiler design, preparing us for the next phase—understanding automata theory and its connection to lexical analysis in the following chapter.

Lighten the bulb!

- Why are keywords typically stored as a list instead of being matched by regular expressions?

4.6 Programming Problems

P 4.1: Write a program that accepts input for RE (ab)^2+.

The input must contain at least two occurrences of "ab" (abab as the minimum valid input) and can be repeated any number of times (abababab, etc.). Use a loop to check the pattern "ab" appears or not.

```
#include <stdio.h>
#include <string.h>
bool isRE(const char *str) {
    int len = strlen(str);
    if (len < 4 || len % 4 != 0)
      return false;
    for (int i = 0; i < len; i += 2) {
        if (str[i] != 'a' || str[i + 1] != 'b') {
            return false;
        }
    }
    return true;
}
int main() {
    char input[100];
    printf("Enter a string: ");
    scanf("%s", input);
    if (isRE(input))
    { printf("Accepted\n"); }
    else { printf("Rejected\n"); }

    return 0;
}
```

Input	Output
ababab abababab	Rejected Accepted

P 4.2: Write a program to accept input for the regular expression (01|10)+.

Strings containing at least one occurrence of "01" or "10", repeating any number of times.

```
#include <stdio.h>
#include <string.h>
bool isValidPattern(char *str) {
    int len = strlen(str);
    if (len < 2) return false;

    for (int i = 0; i < len; i += 2) {
       if (!( (str[i] == '0' && str[i + 1] == '1') || (str[i] ==
'1' && str[i + 1] == '0') ) )
            return false;
    }
       return true;
}
int main() {
    char input[100];

    printf("Enter a string: ");
    scanf("%s", input);

    if (isValidPattern(input)) {printf("Valid string\n");}
    else {printf("Invalid string\n");}

    return 0;
}
```

Input	Output
01 01100110 010	Valid string Valid string Invalid string

P 4.3: Write a program to accept input for the regular expression (a|b)+c.

A string containing at least one occurrence of "a" or "b", ending with "c".

```
#include <stdio.h>
#include <string.h>
bool isValidPattern(char *str) {
    int len = strlen(str);

    if (len < 2) return false;

    for (int i = 0; i < len - 1; i++) {
        if (str[i] != 'a' && str[i] != 'b')
            return false; }
    return str[len - 1] == 'c';  // Last character be 'c'
}
int main() {
    char input[100];

    printf("Enter a string: ");
    scanf("%s", input);

    if (isValidPattern(input)) {printf("Valid string\n");}
    else { printf("Invalid string\n"); }

    return 0;
}
```

Input	Output
ac 123c abcx bbbac bac	Valid string Invalid string Invalid string Valid string Valid string

Exercises

Conceptual Questions

1. Write the regular expression of Set of all strings having even number of aa and even number of bb.
2. Write a valid regular expression of an URL address or email address..
3. What are the languages of this regular expressions?
   a. (ab*)+c?
   b. 1*0+1?0
   c. (ab*)bb(ab)+
   d. aa(bc)*cc
   e. abc(a+b+c+)abc*
4. What are the regular expressions of these languages?
   a. All strings generated by 0*(10*)*. So they are equivalent
   b. All strings of a's and b's that do not contain the subsequence abb.
   c. All strings of a's and b's that contain at most two b's.
   d. All strings of a's and b's with an even number of a's.

Programming Problems

1. Write a program to accept input for the RE (abc)*.
2. Write a program to accept input for the RE b(a|b)*b.
3. Write a program to check if a given string follows the RE (0|1)*1(0|1) (any binary string ending in "1" followed by one more bit).

Explore the codes in **GitHub**

Click or Scan the QR

5

Finite Automata – NFA & DFA

In the previous chapter, we explored Regular Expressions (REs) and their role in defining token patterns. However, to practically recognize these patterns, we need a mechanism that processes input and determines whether it matches a given RE. This is where Finite Automata (FA) come into play.

Finite Automata are abstract mathematical models that define how a sequence of symbols is processed and whether it belongs to a particular language. They are widely used in Lexical Analysis, where the compiler needs to recognize identifiers, numbers, keywords, and other tokens in the source code.

In this chapter, we will first understand the formal definition of Finite Automata, represented using a five-tuple (Q, ∑, δ, q0, F). We will then explore the two primary types of finite automata: Nondeterministic Finite Automata (NFA) and Deterministic Finite Automata (DFA), along with their graphical representations. Finally, we will compare DFA and NFA, discussing their efficiency and application in lexical analysis.

5.1 Finite Automata: An Overview

A Finite Automaton (plural: Finite Automata) is a mathematical model of computation used for recognizing patterns in a sequence of symbols. It consists of a finite number of states and transitions between them based on input symbols.

To illustrate, consider the task of recognizing a valid identifier in a programming language. The automaton would begin in an initial state, move to another state if it encounters a letter, continue moving if more letters or digits appear, and finally reach an accepting state when the end of the identifier is reached. If an invalid character appears, the automaton will reject the string.

A Finite Automaton (FA) is formally defined as a five-tuple:

$$(Q, \Sigma, \delta, q0, F)$$

Where:

Q: A finite set of states.

∑ (Sigma): A finite set of input symbols (alphabet).

δ (Delta): A transition function that maps a state and input symbol to another state.

q0: The initial state (one of the states in Q).

F: A set of final (accepting) states, a subset of Q

Consider a Finite Automaton that recognizes valid identifiers (starting with a letter, followed by letters or digits).

States (Q) = {q0, q1}

Alphabet (∑) = {A-Z, a-z, 0-9}

Transition Function (δ) = q0 → q1 , q1 → q1

Start State (q0) = q0

Final State (F) = {q1}

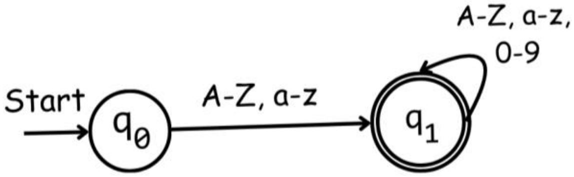


Figure 5.1 Finite Automaton for Identifiers

Types of Finite Automata

Finite Automata is divided into 2 types that is Deterministic Finite Automaton (DFA) and Non-deterministic Finite Automaton (NDFA / NFA).

5.2 Nondeterministic Finite Automata (NFA)

A Nondeterministic Finite Automaton (NFA) is a type of automaton where multiple possible transitions can exist for the same input symbol. Unlike deterministic models, an NFA can be in multiple states at the same time, meaning that a single input may lead to more than one possible path.

Another unique feature of an NFA is the presence of ε (epsilon) transitions, which allow the automaton to change states without consuming any input character. This feature makes NFAs more flexible but also more complex to implement in a real compiler.

For example, an NFA of regular expression **`(a | b)* abb`** is given below along with the 5 tuples of it.

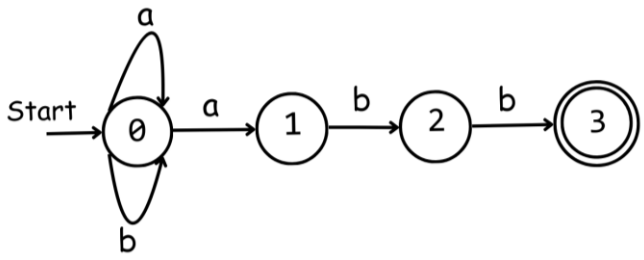


Figure 5.2 NFA of (a | b)* abb

Here the 5 tuples are :

States (Q) = {0,1,2,3}

Alphabet (∑) = {a,b}

Start State (q0) = {0}

Final State (F) = {3}

State	Input Symbol	
	a	**b**
→ 0	{0,1}	{0}
1	∅	{2}
2	∅	{3}

Table I : Transition Table

Transition Function (δ) , a transition table is used for better understanding in general, as shown in Table I. Here ∅ is used if there is no transition of input symbol in a state.

5.3 Deterministic Finite Automata (NFA)

A Deterministic Finite Automaton (DFA) is a refined version of an automaton where each input symbol leads to a single, well-defined state transition. Unlike an NFA, where multiple paths may exist for a given input, a DFA ensures that for every symbol read, the automaton moves deterministically to exactly one next state. This property makes DFAs predictable, structured, and more suitable for implementation in compilers.

One of the defining characteristics of a DFA is the absence of ε-transitions. Every transition in a DFA is explicitly dictated by an input character, ensuring that every move in the automaton consumes an input symbol. This rigid structure enables a DFA to process input sequences in a straightforward and efficient manner, eliminating ambiguity and reducing the need for backtracking.

For example, a DFA of a regular expression is given below along with the 5 tuples of it.

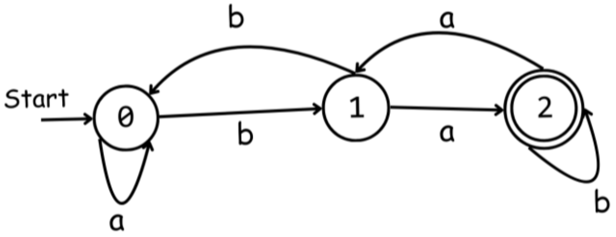


Figure 5.3 An example of DFA

Here the 5 tuples are :

States (Q) = {0,1,2}

Alphabet (∑) = {a, b}

Start State (q0) = {0}

Final State (F) = {2}

State	Input Symbol	
	a	**b**
→ 0	{0}	{1}
1	{2}	{0}
2	{1}	{2}

Table II : Transition Table

Transition Function (δ) , a transition table is used for better understanding in general, as shown in Table II. Here ∅ is used if there is no transition of input symbol in a state.

5.4 Comparing DFA and NFA

While both DFA and NFA can recognize the same languages, their structure and efficiency differ significantly. The key distinctions are:

6. **Determinism**: DFA has only one transition per input, while NFA can have multiple transitions for the same input.
7. **Empty String (ε):** DFA does not allow empty (ε) transitions, but NFA permits them.
8. **Backtracking**: DFA follows a single path without backtracking, whereas NFA may explore multiple paths.
9. **Space Efficiency**: DFA requires more states and memory, while NFA is more compact.
10. **Acceptance**: DFA accepts a string only if one unique path reaches a final state, whereas NFA accepts if at least one path leads to a final state.

5.5 Conversion from NFA to DFA

Since NFAs allow multiple transitions for the same input, they are not directly implementable in compilers. To make them usable for lexical analysis, we need to convert an NFA into an equivalent DFA, where each input leads to a single unique transition. This is done using the Subset Construction Method, which groups sets of NFA states into single DFA states.

Understanding the Subset Construction Method

- For every state in the NFA, determine all reachable states for each input symbol.
- These sets of reachable states **form the new states** in the DFA.
- The **start state** of the DFA remains the same as in the NFA.
- Any state in the DFA that contains an **NFA final state** will be marked as a **DFA final state**.

Knowing the Algorithm

Input: An NFA defined by its states, transitions, and final states. Output: An equivalent DFA that recognizes the same language.

1. Construct the state table for the given NFA, listing all transitions for each state under each input symbol.
2. Initialize an empty state table for the DFA with columns for each input symbol.
3. Mark the start state of the DFA as the same as the NFA's start state.
4. For each new DFA state created, determine its transitions by combining the reachable NFA states for each input symbol.
5. Repeat Step 4 for all newly created DFA states, ensuring that all possible state transitions are accounted for.
6. Any DFA state that contains an NFA final state becomes a final state in the DFA.

This algorithm guarantees that the resulting DFA accepts the same language as the original NFA but follows a single deterministic path for any input string. Now, let's look at a step-by-step example to demonstrate this process using a state transition table and graphical representation.

Let’s consider an NFA that is given below in figure 5.4.

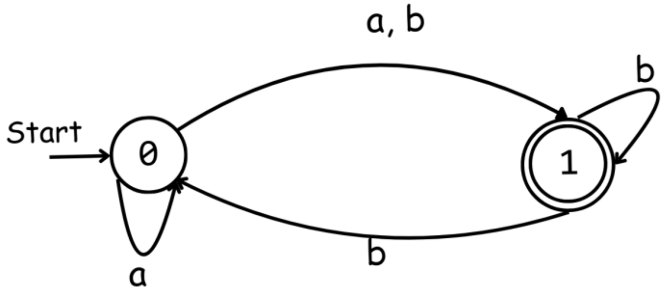


Figure 5.4 NFA

The transition table of this Fig 5.4 is given in Table III, from where we will create a subset table starting with the initial state.

State	Input Symbol	
	a	**b**
→ 0	{0,1}	{1}
1	∅	{0,1}

Table III : Transition Table

Now a subset table will be made from the transition table following the Subset Construction algorithm.

State	Input Symbol	
	a	**b**
→ 0	{0,1}	{1}
1	∅	{0,1}
→ 0,1	{0,1}	{0,1}

Table IV : Subset Table

From this subset table now the DFA will be drawn maintain the basic rules, see fig. 5.5.

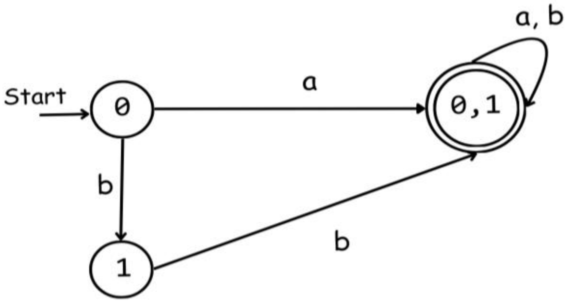


Figure 5.5 DFA

This follows all the constraints of DFA and in this way we can do conversion of NFA to DFA, for better understanding do practice.

Lighten the bulb!

- Why do we say that a DFA is a special case of NFA?

5.6 Programming Problems

P 5.1: Write a program that does tokenization.

Read the string and ignoring the whitespace, print all the words in new line.

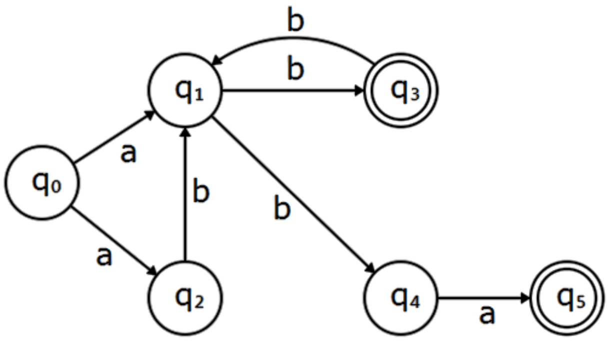
```c
#include<stdio.h>
#include<string.h>
int main()
{
    char str[100], delim[] = " ,.";
    char *token;

    printf("Enter a string: ");
    fgets(str, sizeof(str), stdin);

    str[strcspn(str, "\n")] = '\0';
       // Remove newline character

    token = strtok(str, delim);

    while (token != NULL) {
        printf("%s\n", token);
        token = strtok(NULL, delim);
    }
 return 0;
}
```

Input	Output
int x = 10 + y;	Int x = 10 + Y ;

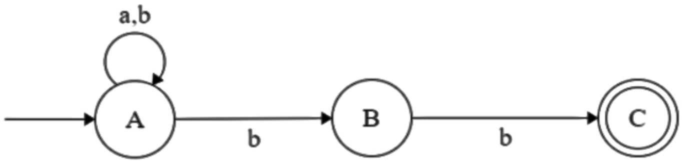
Exercises

Conceptual Questions

1. What is a finite automaton?
2. Write the difference between NFA and DFA.
3. Convert the following NFA to DFA.



4. Convert the following NFA to DFA



Programming Problems

1. Write a program that creates a Syntax table (letter, digits, symbol, arithmetic/logical op….).
2. Write a program that does tokenization without strtok() function.

Explore the codes in **GitHub**
Click or Scan the QR

6

Left Recursion & Left Factoring

When parsing a program, certain grammar structures can create issues for top-down parsers. Left recursion occurs when a non-terminal refers to itself at the beginning of a production, leading to infinite recursion in recursive descent parsing. To avoid this, we must eliminate left recursion.

Left factoring is needed when multiple productions share a common prefix, making it difficult for the parser to decide which rule to apply. By restructuring the grammar, left factoring ensures smooth predictive parsing. In this chapter, we will learn how to identify and remove left recursion and apply left factoring to improve grammar structure.

6.1 Introduction to Left Recursion

In a context-free grammar (CFG), if a production has a non-terminal on the left-hand side that recursively appears at the beginning of its own right-hand side, it is called left recursion. This form of recursion can cause issues during parsing and must be eliminated to avoid infinite loops in recursive descent parsers.

Consider the production:

$$\boldsymbol{A \rightarrow A\alpha \mid \beta}$$

where A is a non-terminal, α is a sequence of terminals and non-terminals, and β is any string that does not start with A. In this case, when we attempt to derive strings, we keep substituting A with Aα, leading to infinite recursion. This prevents the parser from making progress in deriving valid sentences.

6.2 Eliminating Left Recursion

To eliminate left recursion, we transform a left-recursive grammar into an equivalent grammar that generates the same language but does not have left-recursive rules.

For a production rule of the form:

$$\boldsymbol{A \rightarrow A\alpha \mid \beta}$$

where β does not start with A, we can replace it with the following equivalent rules:

$$\boldsymbol{A \rightarrow \beta A'}$$

$$\boldsymbol{A' \rightarrow \alpha A' \mid \epsilon}$$

where **A' (A prime)** is a new non-terminal that helps remove recursion.

Example of Left Recursion Elimination

Example 6.1

Original Grammar: $\boldsymbol{A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \beta_1 \mid \beta_2}$

Here A is left recursive because it appears at the start of its own production rule.

After Elimination:

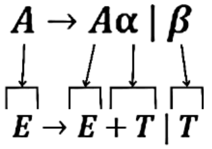
$$\boldsymbol{A \rightarrow \ \beta_1 A' \mid \beta_2 A'}$$
$$\boldsymbol{A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \epsilon}$$

Example 6.2

Original Grammar:

$$\boldsymbol{E \rightarrow E + T \mid T}$$

Here E is left recursive because it appears at the start of its own production rule.


$A \rightarrow A\alpha \mid \beta$
$E \rightarrow E + T \mid T$

Now mapping with the standard format , we can approach to eliminate the left recursion.

After Elimination:

$$\boldsymbol{E \rightarrow TE'}$$

$$\boldsymbol{E' \rightarrow +TE' \mid \epsilon}$$

Example 6.3

Original Grammar:

$$\boldsymbol{A \rightarrow ABd \mid Aa \mid a}$$
$$\boldsymbol{B \rightarrow Be \mid b}$$

Here A and B both are left recursive because it appears at the start of its own production rule.

After Elimination:

$$\boldsymbol{A \rightarrow aA'}$$
$$\boldsymbol{A' \rightarrow BdA' \mid aA' \mid \epsilon}$$
$$\boldsymbol{B \rightarrow bB'}$$
$$\boldsymbol{B' \rightarrow eB' \mid \epsilon}$$

In this way , by mapping with the base structure we can eliminate the left recursion. For more examples and more problems see the exercises.

6.3 Introduction to Left Factoring

A grammar is said to be non-deterministic when multiple productions of a non-terminal share a common prefix. This ambiguity makes predictive parsing difficult.

For example, consider the grammar:

$$\boldsymbol{A} \rightarrow \alpha\mathrm{A} \mid \boldsymbol{\alpha\beta}$$

Here, both productions begin with α, which makes it non-deterministic. To resolve this issue, we apply left factoring to rewrite the grammar in a way that enables the parser to make a deterministic choice.

6.4 Eliminating Left Factoring

To remove left factoring, we factor out the common prefix and introduce a new non-terminal to maintain the same language.

For a non-deterministic grammar of the form:

$$\boldsymbol{A \rightarrow \alpha A \mid \alpha\beta}$$

where **α** is factoring, we can replace it with the following equivalent rules:

$$\boldsymbol{A \rightarrow \alpha A'}$$

$$\boldsymbol{A'} \rightarrow \mathrm{A} \mid \boldsymbol{\beta}$$

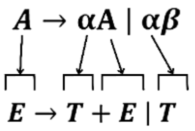
where **A' (A prime)** is a new non-terminal that differentiates the choices after **α**.

Example of Left Factoring Elimination

Example 6.4

Original Grammar: $\boldsymbol{E \rightarrow T + E \mid T}$

Here A is left factoring because here T is a common prefix of production rule.



Now mapping with the standard format , we can approach to eliminate the left factoring.

After Elimination:

$$\boldsymbol{E \rightarrow TE'}$$
$$\boldsymbol{E' \rightarrow +E \mid \epsilon}$$

Example 6.5

Original Grammar:

$$A \rightarrow aAB \mid aA \mid a$$

Here A is left factoring because here a and A is a common prefix of production rule.

After Elimination:

$$\begin{aligned} &A \rightarrow aA' \\ &A' \rightarrow AB \mid A \mid \epsilon \\ &A' \rightarrow AA'' \\ &A'' \rightarrow B \mid \epsilon \end{aligned}$$

Example 6.6

Original Grammar:

$$\begin{aligned} &A \rightarrow aAB \mid aA \\ &B \rightarrow bB \mid b \end{aligned}$$

Here A is left factoring because here aA and b is a common prefix of production rule.

After Elimination:

$$\begin{aligned} &A \rightarrow aAA' \\ &A' \rightarrow B \mid \epsilon \\ &B \rightarrow bB' \\ &B' \rightarrow B \mid \epsilon \end{aligned}$$

In this way , by mapping with the base structure we can eliminate the left factoring. For more examples and more problems see the exercises.

Lighten the bulb!

- Is there a case where left recursion elimination or left factoring might change the language generated by the grammar?

6.5 Programming Problems

P 6.1: Write a program that Identify and Count Articles in a Given String.

Just look for a , an , the in the string making it lowercase.

```c
#include <stdio.h>
#include <string.h>

int main() {
    char str[200], *token;
    int count = 0;

    printf("Enter a sentence: ");
    fgets(str, sizeof(str), stdin);

    token = strtok(str, " ,.-\n");

    while (token != NULL) {
        if (strcmp(token, "a") == 0 || strcmp(token, "an") == 0
|| strcmp(token, "the") == 0)
            count++;
        token = strtok(NULL, " ,.-\n");
    }

    printf("Number of articles: %d\n", count);
    return 0;
}
```

Input	Output
The cat is sitting on a mat with an apple.	Number of articles: 3

Exercises

Conceptual Questions

1. Eliminate the left recursion of the following grammars:

   a) $S \rightarrow Sa |\ Sb \mid c \mid d$

   b) $E \rightarrow E + E \mid ExE \mid a$

   c) $E \rightarrow E + T \mid T\ T \rightarrow T \times F \mid F$
      $T \rightarrow T \times F \mid F$
      $F \rightarrow id$

   d) $S \rightarrow S0S1S \mid 01$

2. Eliminate the left factoring of the following grammars:

   a) $S \rightarrow iEts \mid iEtSes \mid a$
      $E \rightarrow b$

   b) $S \rightarrow bSSaas \mid bSSaSb \mid bSb \mid a$

   c) $A \rightarrow aAB \mid aBc \mid aAc$

   d) $E \rightarrow edu \mid edu.bd \mid education \mid educate \mid educated$

Visit
compiler-forge.netlify.app

Programming Problems

1. Write a program count each article type.
2. Write a program that eliminate extra article in a sentence.

Explore the codes in **GitHub**
Click or Scan the QR

7

First and Follow

Parsing plays a crucial role in syntax analysis, ensuring that a program follows the correct grammatical structure. To facilitate efficient parsing, certain properties of grammar, such as First and Follow sets, help determine which rules to apply when constructing a predictive parser like LL(1) parsing.

The First set of a non-terminal consists of the possible starting terminals of the strings derived from that non-terminal. The Follow set contains the possible terminals that can appear immediately after the non-terminal in some derivation. These sets help in constructing LL(1) parsing tables, enabling efficient top-down parsing without backtracking. In this chapter, we will explore the concepts of First and Follow, their rules, and how they are used to build an LL(1) parsing table for predictive parsing.

7.1 First Set and Its Rules

The First set of a non-terminal A is the set of terminals that appear as the first symbol in any string derived from A. It helps the parser determine which production to apply when expanding A.

Rules to Compute First Set

- If **X** is a terminal, then $\boldsymbol{First(X)} = \{\boldsymbol{X}\}$.
- If **X** is a non-terminal with a production $\boldsymbol{X \rightarrow Y_1 Y_2 \ldots Y_n}$ then:
  - Add $\boldsymbol{First(Y_1)}$ to $\boldsymbol{First(X)}$ (excluding ε if present).

  - If $\boldsymbol{Y_1}$ contains ε, check $\boldsymbol{Y_2}$, and so on.

  - If all $\boldsymbol{Y_i}$ contain ε, then add ε to $\boldsymbol{First(X)}$.
- If **X** derives ε (empty string), add ε to $\boldsymbol{First(X)}$.

For example, in case of the grammar,

$$S \rightarrow AB A \rightarrow a \mid \varepsilon B \rightarrow b \mid c$$

$$A \rightarrow a \mid \varepsilon B \rightarrow b \mid c$$

$$B \rightarrow b \mid c$$

The first set will be :

Non-terminal	First
S	{a, b, c}
A	{ a, ε }
B	{ b, c }

In case of finding the first set, it is better to start from the bottom, this will help finding the set faster and in an easier way.

7.2 Follow Set and Its Rules

The Follow set of a non-terminal A consists of all terminals that can appear immediately after A in a derivation. It is useful for predictive parsing, particularly in LL(1) parsing tables.

Rules to Compute First Set

- **Start Symbol Rule**: Always add $ (end-of-input symbol) to Follow (start symbol).
- If $\boldsymbol{A \to pBq}$ is a production, where p, B and q are any grammar symbols, then everything in $\boldsymbol{First(q)}$ except **ε** is in $\boldsymbol{Follow(B)}$.
- If $\boldsymbol{A \to pB}$ is a production, then everything in $\boldsymbol{Follow(A)}$ is in $\boldsymbol{Follow(B)}$.
- If $\boldsymbol{A \to pBq}$ is a production, and $\boldsymbol{First(q)}$contains **ε** , then $\boldsymbol{Follow(B)}$ contains { $\boldsymbol{First(q) - \varepsilon}$ } ∪ $\boldsymbol{Follow(A)}$.

For example, in case of the grammar,

$$S \to AB$$
$$A \to a \mid \varepsilon B \to b \mid c$$
$$B \to b \mid c$$

The follow set will be :

Non-terminal	Follow
S	{ $ }
A	{ b, c }
B	{ $ }

In case of finding the follow set, first construct the first set and then, the follow set.

7.3 LL(1) Parsing and Construction of Parsing Table

LL(1) parsing is a top-down predictive parsing technique where the first L stands for Left-to-right scanning, the second L for Leftmost derivation, and (1) means it uses one lookahead symbol to make parsing decisions.

Steps to Construct an LL(1) Parsing Table

- For each production $\boldsymbol{A \rightarrow \alpha}$, add $\boldsymbol{\alpha}$ in $\boldsymbol{Table}[\boldsymbol{A}, \boldsymbol{X}]$ for each $\boldsymbol{X} \in \boldsymbol{First}(\boldsymbol{\alpha})$.
- If $\boldsymbol{\varepsilon}$ is in $\boldsymbol{First}(\boldsymbol{\alpha})$, add $\boldsymbol{A \rightarrow \alpha}$ in $\boldsymbol{Table}[\boldsymbol{A}, \boldsymbol{X}]$ for each $\boldsymbol{X} \in \boldsymbol{Follow}(\boldsymbol{A})$.
- If there is a conflict (multiple entries in one cell), the grammar is not LL(1).

Example Parsing Table of the previously solved grammar in First and Follow is:

Non-Terminal	a	b	c	$
S	$S \rightarrow AB$	$S \rightarrow AB$	$S \rightarrow AB$	
A	$A \rightarrow a$	$A \rightarrow \varepsilon$	$A \rightarrow \varepsilon$	
B		$B \rightarrow b$	$B \rightarrow c$	

In the table Rows will contain the non-Terminals and the column will contain the Terminal Symbols. All the Null Productions of the Grammars will go under the Follow elements and the remaining productions will lie under the elements of First set.

Now for better understanding another example is given for the production rule :

$$E \rightarrow TE'$$
$$E' \rightarrow +T\,E' \mid \varepsilon$$
$$\mathrm{T} \rightarrow \mathrm{FT}'T' \rightarrow *FT' \mid \varepsilon\mathrm{F} \rightarrow \mathrm{id} \mid (\,\mathrm{E}\,)$$
$$T' \rightarrow *FT' \mid \varepsilon\mathrm{F} \rightarrow \mathrm{id} \mid (\,\mathrm{E}\,)$$
$$\mathrm{F} \rightarrow \mathrm{id} \mid (\,\mathrm{E}\,)$$

The first and follow set of the above grammar is given in the following table:

Non-terminal	First	Follow
E	{ (, id }	{ $,) }
E'	{ +, ε }	{ $,) }
T	{ (, id }	{ + , $,) }
T'	{ *, ε }	{ + , $,) }
F	{ id, (}	{ *, +, $,) }

Now based on the first and follow sets the LL(1) parsing table is constructed below:

Non-Terminal	+	*	id	(	)	$
E			$E \rightarrow TE'$	$E \rightarrow TE'$		
E'	$E' \rightarrow +TE'$				$E' \rightarrow ε$	$E' \rightarrow ε$
T			T → FT′	T → FT′		
T'	$T' \rightarrow ε$	$T' \rightarrow * FT'$			$T' \rightarrow ε$	$T' \rightarrow ε$
F			F → id	F → (E)		

With First and Follow sets and the LL(1) parsing table, we can analyze and parse context-free languages effectively. The next step is to understand shift-reduce parsing, which is another parsing technique used in bottom-up parsers.

Lighten the bulb!

- Can a left-recursive grammar be LL(1)? Why or why not?

7.4 Programming Problems

P 7.1: Write a program that identify and count the number of each vowel.

Just look for a , an , the in the string making it lowercase.

```c
#include <stdio.h>
#include <ctype.h>

int main() {
    char str[100];
    int count[5] = {0}; // a, e, i, o, u

    printf("Enter a string: ");
    scanf("%99[^\n]", str);

    for (int i = 0; str[i] != '\0'; i++) {
        char ch = tolower(str[i]);
        switch (ch) {
            case 'a':
                count[0]++;
                break;
            case 'e':
                count[1]++;
                break;
            case 'i':
                count[2]++;
                break;
            case 'o':
                count[3]++;
                break;
            case 'u':
                count[4]++;
                break;
        }
    }

    printf("Number of vowels:\n");
    printf("a: %d\n", count[0]);
    printf("e: %d\n", count[1]);
    printf("i: %d\n", count[2]);
    printf("o: %d\n", count[3]);
    printf("u: %d\n", count[4]);

    return 0;
}
```

Input	Output
Enter a string: This is an Example sentence	Number of vowels: a: 2 e: 4 i: 3 o: 0 u: 0

Exercises

Conceptual Questions

1. Find sets of FIRST & FOLLOW and construct LL(1) parse table for the following grammar:

   a) S → Bb | cdB → aB | ε
   B → aB | ε
   C → cC | ε

   b) P → SQTRQQ → SR | ce |ε
   Q → SR | ce |ε
   R → n | – | ε
   S → T | hellT → fra | 0

   T → fra | 0

   c) S → ACB|Cbb|Ba
   A → da|BC
   B → g | ε
   C → h | ε

Visit
compiler-forge.netlify.app

Programming Problems

1. Write a program to identify and count vowel and consonant
2. Write a program to extract and print only vowels from a sentence.
3. Find the word with the most vowels in a sentence.

Explore the codes in **GitHub**
Click or Scan the QR

8

LR0 Parser and Canonical Table

Parsing techniques are essential in compiler design to analyze the structure of a given input based on grammar rules. While LL(1) parsing follows a top-down approach, LR(0) parsing is a bottom-up approach that efficiently handles a larger class of grammars. LR(0) parsers use shift-reduce parsing and work by constructing a canonical collection of LR(0) items to determine parsing actions systematically.

In this chapter, we will explore LR(0) parsing, its rules, and how to construct an LR(0) parsing table using the canonical collection of items.

8.1 LR(0) Parsing and Its Basic Rules

An LR(0) parser processes input using a bottom-up shift-reduce approach without lookahead symbols. It constructs a canonical collection of LR(0) items to determine parsing actions systematically. The parsing process follows these essential rules:

1. **Add Augmented Production**: Introduce an augmented production $S' \rightarrow .S^0$ and insert the **dot (•)** at the first position of every production along superscript numbering starting with 0.
2. **Compute Closure for $\mathbf{I_0}$ State**: Start with the augmented production in **state $\mathbf{I_0}$** and compute the **closure** by adding all possible productions with the **dot (•) at the beginning**.
3. **Compute GOTO for Each Symbol**: For every state, apply the **GOTO function** on grammar symbols (terminals & non-terminals) by shifting the dot **one position ahead**.
4. **Repeat Closure & GOTO Until Completion**: Keep applying **closure and GOTO** iteratively until no new states are generated.
5. Construct the Parsing Table: Use the generated canonical LR(0) states to create the ACTION and GOTO table, determining shift, reduce, accept, or error actions.

This systematic approach ensures an efficient and deterministic parsing mechanism using LR(0) parsing.

LR(0) Parsing Actions

- **Shift(S)**: Move the next input symbol onto the stack and transition to a new state.
- **Reduce(r)**: Replace a **right-hand side** of a production with its **left-hand side** in the stack.
- **Accept**: If the input is fully parsed and reaches the start symbol's reduction, accept it.

Canonical Collection / Table of LR(0) Items

To construct an LR(0) parsing table, we generate a canonical collection of LR(0) items using the closure and goto functions.

Rules of constructing canonical table:

- If a state is going to some other state on a terminal, then it corresponds to a shift move.
- If a state is going to some other state on a variable, then it corresponds to go to move.
- If a state contains the final item in the particular row, then write the reduce node completely.

A combined example is given in the next page for better understanding of this concept.

Example of LR(0) Parsing

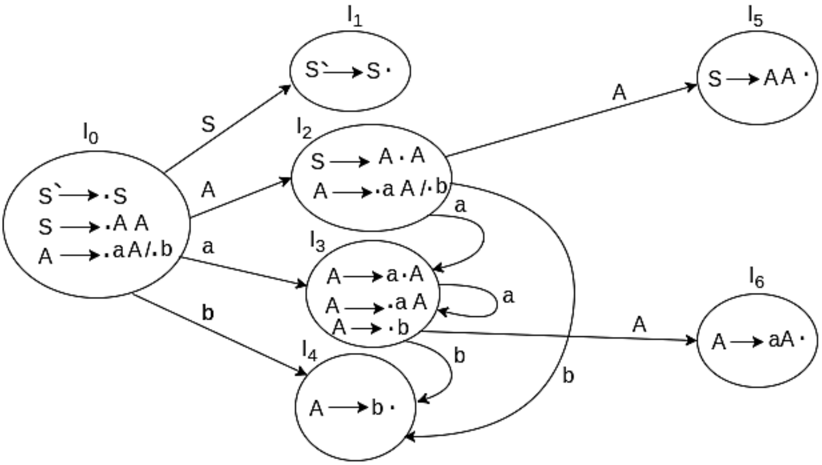
Consider the grammar:

$$S \to AA$$
$$A \to aA \mid b$$

The augmented grammar of this is :

$$S' \to .S \ ^0$$
$$S \to .AA \ ^1$$
$$A \to .aA \ ^2$$
$$A \to .b \ ^3$$

Then the LR(0) parsing:



Here, **I$_0$**: Initial state with **augmented production** and closure applied.

I$_1$: Goto (I$_0$, S) → Accept state.

I$_2$: Goto (I$_0$, A) → Expand with productions of A.

I$_3$: Goto (I$_2$, a) and I$_0$, a) → Expand A → aA.

I$_4$: Goto (I$_2$, b) and I$_0$, b) → Closure on A → b.

I$_5$: Goto (I$_2$, A) → Shift dot in S → AA.

I$_6$: Goto (I$_3$, A) → Shift dot in A → aA.

Canonical Table:

States	**Actions**			**GOTO**	
	a	**b**	**$**	**S**	**A**
I_0	S_3	S_4		1	2
I_1			Accepted		
I_2	S_3	S_4			5
I_3		S_4			6
I_4	r_3	r_3	r_3		
I_5	r_1	r_1	r_1		
I_6	r_2	r_2	r_2		

Explanation:

- I_0 on S is going to I_1 so write it as 1.
- I_0 on A is going to I_2 so similarly write it as 2.
- I_2 on A is going to I_5 so similarly write it as 5.
- I_3 on A is going to I_6 so similarly write it as 6.
- I_0, I_2 and I_3 on **a** are going to I_3 so write it as **S_3** which means that shift 3.
- I_0, I_2 and I_3 on **b** are going to I_4 so write it as **S_4** which means that shift 4.
- I_4, I_5 and I_6 all states contain the final item because they contain • (dot) in the right most end. So, rate the production as production number.

Lighten the bulb!

- How does the Augmented Grammar help in LR(0) parsing?

8.2 Programming Problems

P 8.1: Write a C program to recognize a set of identifiers valid or not.

Check if the first character is an alphabet (a-z, A-Z) or an underscore (_). Traverse the remaining characters and ensure they consist only of alphabets, digits (0-9), or underscores (_).

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>

int isValidIdentifier(char *identifier) {

    if (!isalpha(identifier[0]) && identifier[0] != '_') {
        return 0;
    }
    for (int i = 1; identifier[i] != '\0'; i++) {
        if (!isalnum(identifier[i]) && identifier[i] != '_') {
            return 0;
        }
    }
    return 1;
}

int main() {
    int n;

    printf("Enter the no. of identifiers: ");
    scanf("%d", &n);

    char identifiers[n][100];

    printf("Enter identifiers:\n");
```

```c
    for (int i = 0; i < n; i++) {
        scanf("%s", identifiers[i]);
    }

    printf("\nResults:\n");

    for (int i = 0; i < n; i++) {
        if (isValidIdentifier(identifiers[i])) {
            printf("%s - Valid\n", identifiers[i]);
        }

        else {
            printf("%s - Invalid\n", identifiers[i]);
        }
    }

    return 0;
}
```

Input	Output
Enter the no. of identifiers: 5 Enter identifiers: _var123 2start myVar valid_name $invalid	Results: _var123 - Valid 2start - Invalid myVar - Valid valid_name - Valid $invalid - Invalid

Exercises

Conceptual Questions

1. Construct LR(0) parsing and Canonical Table for the following grammar:

   a. $S \rightarrow A\$A \rightarrow B \mid A * B$
      $A \rightarrow B \mid A * B$
      $B \rightarrow x \mid (A)$

   b. $E \rightarrow E + T \mid TT \rightarrow T * F \mid F$
      $T \rightarrow T * F \mid F$
      $F \rightarrow (E) \mid id$

   c. $S \rightarrow aAd \mid bBd \mid aBe \mid bAeA \rightarrow c$
      $A \rightarrow c$
      $B \rightarrow c$

   d) $S \rightarrow aTReT \rightarrow Tbc \mid b$
      $T \rightarrow Tbc \mid b$
      $R \rightarrow d$

Programming Problems

1. Write a program to check if it's a keyword or not.
2. Write a program to count valid and invalid identifiers.
3. Write a program to find the longest and shortest valid identifiers.

Explore the codes in **GitHub**
Click or Scan the QR

9

Intermediate Code Generation

Intermediate Code Generation serves as a crucial phase in the compilation process, transforming high-level source code into an intermediate representation before generating machine code. The primary goal of this phase is to create a structure that is easy to manipulate while preserving the original program logic.

In this chapter, we will explore syntax trees, directed acyclic graphs (DAGs), and three-address code (TAC). We will also discuss different three-address code representations such as quadruples, triples, and indirect triples, which are commonly used in compiler design.

Intermediate code is used to translate the source code into the machine code. Intermediate code lies between the high-level language and the machine language.

A compiler can generate a middle-level language code, known as intermediate code or intermediate text, during the translation of a source program into an object code for a target machine. Between the source language code and the object code lies the ambiguity of this code. Intermediate code can be represented as a postfix notation, a syntax tree, a directed acyclic graph (DAG), a three-address code, a quadruple, a triple.

9.1 Syntax Tree and Parse Tree

A parse tree represents the syntactic structure of a source program according to a given grammar. However, parse trees are often large and contain redundant information. To simplify representation, a syntax tree (also known as an abstract syntax tree) is used.

Difference between Syntax Tree and Parse Tree

Parse Tree	Syntax Tree
Parse tree is a graphical representation of the replacement process in a derivation.	Syntax tree is the compact form of a parse tree.
Each interior node represents a grammar rule. Each leaf node represents a terminal.	Each interior node represents an operator. Each leaf node represents an operand.
Parse trees provide every characteristic information from the real syntax.	Syntax trees do not provide every characteristic information from the real syntax.
Parse trees are comparatively less dense than syntax trees.	Syntax trees are comparatively denser than parse trees.

Example of a syntax tree and parse tree for the following grammar:

Grammar:

$$\boldsymbol{E \rightarrow E + T \mid T}$$
$$\boldsymbol{T \rightarrow T * F \mid F}$$
$$\boldsymbol{F \rightarrow (E) \mid id}$$

String :

$$\boldsymbol{W = id + id * id}$$

Parse Tree for the given string,

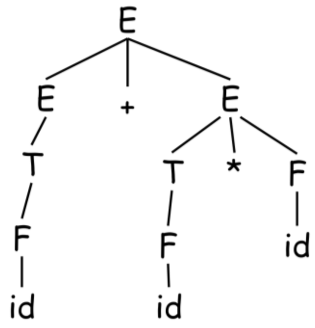


Figure 9.1 Parse Tree

Syntax Tree for the given string,

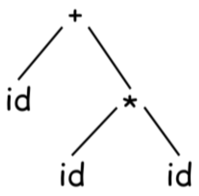


Figure 9.2 Syntax Tree

9.2 Directed Acyclic Graph (DAG)

A DAG is a compressed form of the syntax tree that avoids duplicate computations by reusing common subexpressions.

Rules for Constructing a DAG:

1. Each unique subexpression is represented only once.
2. Nodes represent operations and operands.
3. Common subexpressions share the same node.
4. Leaf nodes represent identifiers or constants.

Example of DAF of the previous page grammar :

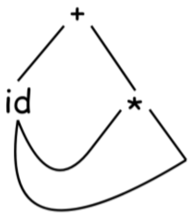


Figure 9.3 DAG

Some basic DAF for better understanding :

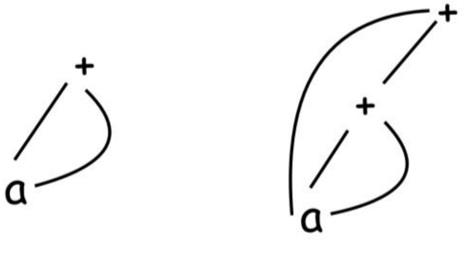


Figure 9.4 DAG of $a + a$ and $a + a + a$

DAF of expression :

$$(((a + a) + (a + a)) + ((a + a) + (a + a)))$$

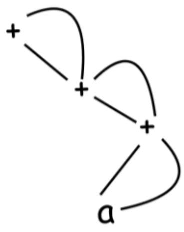


Figure 9.5 DAG

DAF of expression $(a + b) * (a + b + c)$

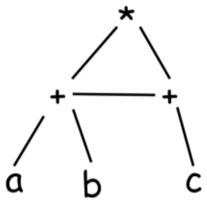


Figure 9.6 DAG

9.3 Three Address Code (TAC)

Three-address code is an intermediate representation that uses at most **three operands** in each instruction. It simplifies complex expressions into a sequence of simple operations.

Common Three-Address Instruction Forms

1. **Assignment:** x = y op z (binary operation)
2. **Unary Operation:** x = op y (e.g., x = -y)
3. Copy Statement: x = y
4. **Conditional Jumps:** if x relop y goto L
5. Unconditional Jumps: goto L
6. Function Calls and Returns: param x, call p, return y

Generation of Three-Address Code

3 address code will be generated based on operator precedence.

Generate the 3-address code for : **`a = b + c + d`**

3-address code:

```
T1 = b + c
T2 = T1 + d
a = T2
```

Generate the 3-address code for :

```
-( a * b ) + ( c + d ) - ( a + b + c + d )
```

3-address code:

```
T1 = a * b
T2 = uminus T1
T3 = c + d
T4 = T1 + T3
T5 = a + b
T6 = T3 + T5
T7 = T4 - T6
```

Generate the 3-address code for :

```
if A < B
then 1
else 0
```

3-address code:

```
(1)   If (A<B)  goto (4)
(2)   T1 = 0
(3)   goto (5)
(4)   T1 = 1
(5)
```

9.4 Implementation of Three-Address Code

Three-address code can be implemented using quadruples, triples, and indirect triples.

Quadruples

It is structure with consist of 4 fields namely op, arg1, arg2 and result. op denotes the operator and arg1 and arg2 denotes the two operands and result is used to store the result of the expression.

Triples

This representation doesn't make use of extra temporary variable to represent a single operation instead when a reference to another triple's value is needed, a pointer to that triple is used. So, it consists of only three fields namely op, arg1 and arg2.

Indirect Triples

This representation makes use of pointer to the listing of all references to computations which is made separately and stored. Its similar in utility as compared to triples representation but requires less space than it. Temporaries are implicit and easier to rearrange code.

Example of Quadruples, Triples and Indirect Triples

Expression: `(x + y) * (y + z) + (x + y + z)`

Three address code :

```
t1 = x + y
t2 = y + z
t3 = t1 * t2
t4 = t1 + z
t5 = t3 + t4
```

Quadruple :

Location	Op	Arg1	Arg2	Result
(1)	+	x	y	t1
(2)	+	y	z	t2
(3)	*	t1	t2	t3
(4)	+	t1	z	t4
(5)	+	t3	t4	t5

Triples :

Location	Op	Arg1	Arg2
(1)	+	x	y
(2)	+	y	z
(3)	*	(1)	(2)
(4)	+	(1)	z
(5)	+	(3)	(4)

Indirect Triples :

	Statement
35	(1)
36	(2)
37	(3)
38	(4)
39	(5)

Location	Op	Arg1	Arg2
(1)	+	x	y
(2)	+	y	z
(3)	*	(1)	(2)
(4)	+	(1)	z
(5)	+	(3)	(4)

Precedence Table of Common Operators

Category	Operator	Associativity
Postfix	( ) [ ] ↑	Left to right
Unary	+ - ! ~ ++ - -	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Equality	== !=	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Assignment	=+ =- =* =/	Right to left

Within an expression, higher precedence operators will be evaluated first.

Lighten the bulb!

- Why is BODMAS not directly used for operator precedence in compilers?

9.5 Programming Problems

P 9.1: Write a C program that will extract Preposition from a given string.

Uses strtok() to split the string into words then convert each word to **lowercase** for case-insensitive matching, lastly check if each word is in the predefined **list of prepositions**.

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>

// List of common prepositions
const char *prepositions[] = {
    "in", "on", "at", "by", "with", "about", "against",
"between", "into", "through", "during", "before", "after",
"above", "below", "to", "from", "up", "down", "under", "over",
"between", "among", "of", "for", "since","without", "within",
"upon"
};

const int prepositionCount = sizeof(prepositions) /
sizeof(prepositions[0]);

// Function to check if a word is a preposition
int isPreposition(char *word) {
    for (int i = 0; i < prepositionCount; i++) {
        if (strcmp(word, prepositions[i]) == 0) {
            return 1; // Word is a preposition
        }
    }
    return 0; // Not a preposition
}
```

```
int main() {
    char str[200];
    char word[50];

    printf("Enter a sentence: ");
    fgets(str, sizeof(str), stdin);

    str[strcspn(str, "\n")] = '\0'; // Remove newline character

    printf("Extracted Prepositions: ");
    char *token = strtok(str, " "); // Tokenize words

    while (token != NULL) {
        // Convert to lowercase for case-insensitive comparison
        for (int i = 0; token[i]; i++) {
            word[i] = tolower(token[i]);
            word[i + 1] = '\0';
        }
        if (isPreposition(word)) {
            printf("%s ", word);
        }
        token = strtok(NULL, " ");
    }
    printf("\n");
    return 0;
}
```

Input	Output
Enter a sentence: She sat on the chair with a book in her hand	Extracted Prepositions: on with in

Exercises

Conceptual Questions

1. Write the difference between parse tree and syntax tree.
2. Construct syntax tree for the following arithmetic expressions:

   a. (a + b) * (c – d) + ((e / f) * (a + b))
   b. (a + a) + b * c + (b * c + c) + (d + d + d + d)
   c. b * - c + b * - c
   d. a + a * (b – c) + (b – c) * d
   e. (a + b) * (c + d) + (a + b + c)

3. Construct the DAF for the following expression:

   a. (a + b) x (a + b + c)
   b. a + a + a + a
   c. a + a + a + a + a
   d. a + b * c – d / (b * c)

4. Generate the 3-address code , quadruples, triples and indirect triples for the following expression:

   a. a + b * c – d / (b * c)
   b. (-c * b) + (-c * d)
   c. (x + y) * (y + z) + (x + y + z)
   d. (a x b) + (c + d) – (a + b + c + d)
   e. If A < B and C < D then t = 1 else t = 0
   f. a + b x c / e ↑ f + b x c
   g. a = b * – c + b * – c
   h. a = −b + c ∗ d

Programming Problems

1. Write a program that does preposition Count from a given string.

Explore the codes in **GitHub**

Click or Scan the QR

10

Control Flow Analysis

Control flow analysis helps in understanding a program's execution structure and optimizing it for efficiency. It involves breaking down the flow of execution into basic blocks and representing them in a flow graph. This process aids in error detection, code optimization, and efficient execution.

A basic block is a group of instructions with one entry point and one exit point, meaning execution flows sequentially without interruption. By organizing the code into basic blocks, the compiler can optimize and analyze it efficiently. A flow graph visually represents the execution paths between basic blocks, helping in program analysis. In this chapter, we will explore basic blocks, leader selection rules, and flow graphs, along with the essential rules for constructing flow graphs.

10.1 Basic Blocks

A basic block is a straight-line sequence of instructions where execution enters at the beginning and exits at the end without any branches or jumps. This structure ensures that statements execute sequentially as a unit, making code optimization and flow analysis more efficient.

Partitioning Intermediate Code into Basic Blocks

Partitioning is done by identifying leaders and grouping instructions accordingly. The process follows these rules:

Leader Selection Rules

1. The first 3-address instruction is the leader.
2. Any instruction that is the target of a conditional or unconditional jump is a leader.
3. Any instruction that immediately follows of a conditional or unconditional jump is a leader.

Basic Block Forming Rules

1. All the statements that follow the leader (including the leader) till the next leader appears form one basic block.
2. The block containing the first leader is called as Initial block.

10.2 Flow Graph

Once an intermediate code program is divided into basic blocks, via a flow graph, we represent the flow of control between them. The nodes of the flow graph are the basic blocks.

Rules of Flow Graph

1. Edges represent control flow between blocks.
2. Blocks without jumps link sequentially to the next block.
3. When a goto statement is present, an arrow is drawn to indicate the jump to the corresponding instruction line.

Example of Basic Block and Flow Graph

Consider the following intermediate code:

```
1)  i = 1
2)  j = 1
3)  t1 = 10 * i
4)  t2 = t1 + j
5)  t3 = 8 * t2
6)  t4 = t3 - 88
7)  a [t4] = 0.0
8)  j = j + 1
9)  if j <= 10 goto 3
10) i = i + 1
11) if i <= 10 goto 2
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a [t6] = 1.0
16) i = i + 1
17) if i <= 10 goto 13
```

Selecting the leaders from the 3-address code:

```
1)  i = 1               {L1}
2)  j = 1               {L2}
3)  t1 = 10 * i         {L2}
4)  t2 = t1 + j
5)  t3 = 8 * t2
6)  t4 = t3 - 88
7)  a [t4] = 0.0
8)  j = j + 1
9)  if j <= 10 goto 3
10) i = i + 1           {L3}
11) if i <= 10 goto 2
12) i = 1               {L3}
13) t5 = i - 1          {L2}
14) t6 = 88 * t5
15) a [t6] = 1.0
16) i = i + 1
17) if i <= 10 goto 13
```

Here L1, L2 and L3 denotes leader and the number denotes based on which number rule they are leader. A single instruction can be multiple leaders by multiple rules.

Now constructing the basic block and flow graph:

Basic Block

```
1)  i = 1                    B1
```

```
2)  j = 1                    B2
```

```
3)  t1 = 10 * i
4)  t2 = t1 + j
5)  t3 = 8 * t2
6)  t4 = t3 - 88             B3
7)  a [t4] = 0.0
8)  j = j + 1
9)  if j <= 10 goto 3
```

```
10) i = i + 1
11) if i <= 10 goto 2        B4
```

```
12) i = 1                    B5
```

```
13) t5 = i - 1
14) t6 = 88 * t5
15) a [t6] = 1.0             B6
16) i = i + 1
17) if i <= 10 goto 13
```

Figure 10.1 Basic Block

Flow Graph

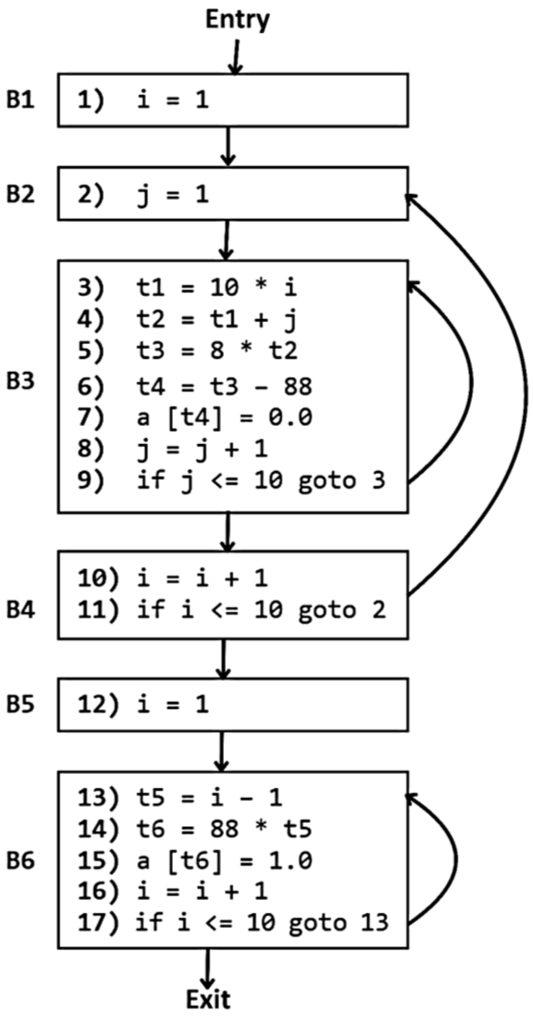


Figure 10.2 Flow Graph

Lighten the bulb!

- Can two different basic blocks share the same instructions?

10.2 Programming Problems

P 10.1: Count and show the max frequency of a word in a string.

Tokenize the input string using delimiters (spaces, punctuation) and store words in an array. Track word frequencies, find the most frequent word, and print the result.

```c
#include <stdio.h>
#include <string.h>
#include <ctype.h>

#define MAX_WORDS 100
#define MAX_WORD_LEN 50

int main() {
    char str[1000], words[MAX_WORDS][MAX_WORD_LEN];
    int freq[MAX_WORDS] = {0}, wordCount = 0, maxFreq = 0;
    char maxWord[MAX_WORD_LEN];

    printf("Enter a string: ");
    fgets(str, sizeof(str), stdin);
    str[strcspn(str, "\n")] = '\0'; // Remove newline character

    for (int i = 0; str[i]; i++) str[i] = tolower(str[i]);
    // Convert to lowercase

    char *token = strtok(str, " ,.-!?");
    while (token) {
        int i, found = 0;
        for (i = 0; i < wordCount; i++) {
            if (strcmp(words[i], token) == 0) {
                freq[i]++;
                found = 1;
                break;
            }
        }
        if (!found) {
            strcpy(words[wordCount], token);
            freq[wordCount++] = 1;
        }
        token = strtok(NULL, " ,.-!?");
    }

    for (int i = 0; i < wordCount; i++) {
        if (freq[i] > maxFreq) {
            maxFreq = freq[i];
            strcpy(maxWord, words[i]);
        }
    }

    printf("Most frequent word: '%s' (Frequency: %d)\n",
maxWord, maxFreq);

    return 0;
}
```

Input	Output
Enter a string: The cat sat on the mat. The mat was clean, and the cat was happy.	Most frequent word: 'the' (Frequency: 4)

Exercises

Conceptual Questions

1. Construct Three Address Code, find & mark leaders, construct basic block and flow graph from the following given code:

```
1) prod : = 0
2) i : = 1
3) t1 : = 4*i
4) t2 : = a[t1]
5) t3 : = 4*i
6) t4 : = b[t3]
7) t5 : = t2*t4
8) t6 : = prod + t5
9) prod : = t6
10) t7 : = I + 1
11) i : = t7
12) if(i <= 20) goto(3)
```

2. Construct Three Address Code, find & mark leaders, construct basic block and flow graph from the following given code:

```
1) sum := 0
2) j := 1
3) t1 := 2 * j
4) t2 := arr[t1]
5) sum := sum + t2
6) if (sum > 30) goto(9)
7) t3 := j + 1
8) j := t3
9) if (j <= 10) goto(3)
10) result := sum
11) if (result < 50) goto(12)
12) final := result
```

Programming Problems

1. Write a program to count and show top 3 max frequency word.
2. Find the Second Most Frequent Word in a given string.
3. Find the Longest Word in a given sentence.

Explore the codes in **GitHub**

Click or Scan the QR

11

Code Optimization

Code optimization is the process of improving the efficiency of a program by reducing execution time, memory usage, or both, without altering its output. Optimized code runs faster and consumes fewer resources, making it crucial for compiler design and software development.

Code optimization can be performed at different levels, High-level optimization: Performed at the source code level. Intermediate code optimization: Applied after code generation but before machine code conversion. Machine code optimization: Specific to processor architecture, reducing redundant operations. Optimizations should maintain correctness, meaning the transformed code must produce the same results as the original.

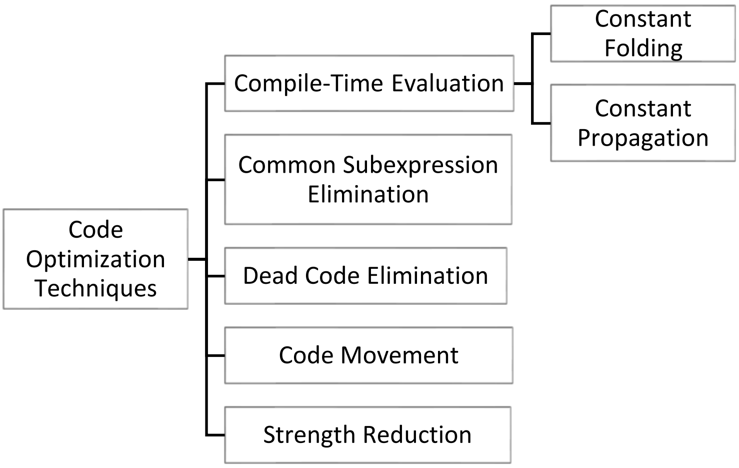
11.1 Why Do We Need Code Optimization?

Code optimization plays a vital role in making programs efficient and scalable.

- Reducing Execution Time: Eliminating unnecessary calculations speeds up program execution.
- Minimizing Memory Usage: Removing redundant variables and expressions reduces memory consumption.
- Enhancing Code Maintainability: Optimized code is often cleaner and easier to understand.
- Improving Compiler Efficiency: Optimizations at the intermediate code level help in generating better machine code.
- Reducing Power Consumption: In embedded systems, optimized code leads to lower power usage.

11.2 Code Optimization Techniques

Several optimization techniques improve program performance. Below are five key methods:



Compile-Time Evaluation

Two techniques that falls under compile time evaluation are-

Constant Folding involves evaluating constant expressions at compile time rather than runtime.

Before:	After Optimization:

x = 10 + 5;
y = x * 2;
	x = 15;
	y = 30;

Here, 10 + 5 is computed at compile time, reducing runtime computation.

Constant Propagation replaces variables with known constant values.

Before:	After Optimization:

x = 5;
y = x * 2;
	y = 10;

Here, since x is always 5, we replace it directly.

Common Subexpression Elimination (CSE)

If an expression is computed multiple times with the same operands, it can be computed once and reused.

Before:	After Optimization:

t1 = a + b;
t2 = a + b;
t3 = t1 * c;
	t1 = a + b;
	t3 = t1 * c;

Here, the redundant computation of a + b is removed.

Dead Code Elimination

Dead code refers to statements that have no impact on the program's final result and can be safely removed.

Before:	**After Optimization:**

a = 5;
b = 10;
c = a + b;
return b;
	return 10;

Here, since c is never used, it is removed, simplifying the code.

Code Movement

This moves calculations that do not change within a loop outside the loop to avoid redundant computation.

Before:	**After Optimization:**

for (i = 0; i < n; i++) {
    t = a * b;
    arr[i] = t + i;
}
	t = a * b;
	for (i = 0; i < n; i++) {
	    arr[i] = t + i;
	}

Here, a * b is constant inside the loop, so it is computed once before the loop.

Strength Reduction

Expensive operations like multiplication and division can be replaced with cheaper operations like addition and bit shifts.

Before:	**After Optimization:**
b = a * 2	b = a + a

Here, Multiplication is replaced with an addition, which is computationally faster.

Lighten the bulb!

- Why do compilers focus on intermediate code optimization rather than direct source code optimization?

11.3 Programming Problems

P 11.1: Write a program that finds the title of a paragraph.

Identify the first period as the marker for title extraction, scanning through the text character by character. Capture all characters from the start of the paragraph up to that first period, while handling any leading whitespace.

```
#include <stdio.h>
#include <ctype.h>
void extractTitle(const char *paragraph, char *title) {
    while (isspace(*paragraph))
     {
      paragraph++;
     } // Skip leading spaces

    while (*paragraph && *paragraph != '.')
      { *title++ = *paragraph++;
        *title = '\0';
      } // Null-terminate
}

int main() {
    char paragraph[500], title[500];
    fgets(paragraph, sizeof(paragraph), stdin);

    extractTitle(paragraph, title);
    printf("Title: %s\n", title);
    return 0;
}
```

Input	Output
The Wonders of Space. Space is vast and full of mysteries waiting to be explored.	Title: The Wonders of Space

P 11.2: Write a program to store multiple strings in One String.

Just take the numbers of string and then store them in a string.

```c
#include <stdio.h>
#include <string.h>

int main() {
    char str[500] = "", temp[100];
    int n;

    printf("Enter number of strings: ");
    scanf("%d", &n);
    getchar();

    for (int i = 0; i < n; i++) {
        printf("Enter string %d: ", i + 1);
        fgets(temp, sizeof(temp), stdin);
        strcat(str, temp);
    }

    printf("\nCombined String:\n%s", str);
    return 0;
}
```

Input	Output
Enter number of strings: 2 Hello World Welcome to C programming	Combined String: Hello World Welcome to C programming

Exercises

Conceptual Questions

1. Why we use Code Optimization?
2. What is the benefit of Code Optimization?
3. Describe all the techniques of Code Optimization with example.

Programming Problems

1. Write a program to count number of para's in a paragraph.
2. Write a program to find multiple title in a string.

Explore the codes in **GitHub**

Click or Scan the QR

Project Spotlight

Compiler Forge Toolkit

The **Compiler Forge Toolkit** is a web-based utility designed to simplify and accelerate learning in compiler design. Built as a companion tool to this book, it allows students and educators to instantly compute critical compiler construction components like Left Recursion Elimination, Left Factoring, and First & Follow Sets — all with just a few clicks.

Whether you're learning the concepts for the first time or building a parser, Compiler Forge helps reduce manual effort and error by providing clean, structured outputs.

Key Features

Struggling with Left Recursion, Left Factoring, or First-Follow calculations? This tool makes it effortless:

- Real-time input/output for grammar-based problems.
- Handles both simple and complex grammar rules.
- Clean UI for easy interaction and copy-paste-ready results.

This tool is particularly helpful for students performing syntax analysis tasks and for instructors looking to demonstrate grammar simplification interactively.

Source Code and Repository

The complete source code for Compiler Forge Toolkit is available on GitHub:
Repository : https://github.com/garodiaa/compiler-forge-toolkit

Feel free to explore, fork, or contribute to the project!

This repository is shared for learning and reference purposes only. Please do not directly copy or submit the code as your own in academic settings. Instead, use it to understand the logic and implement your own version wherever applicable.

Visit
compiler-forge.netlify.app

Project Ideas for Compiler Enthusiasts

If you've enjoyed the concepts and lab programs in this book, here are a few project ideas to take your learning further:

- **NFA/DFA Simulator :** Create a visual simulator that converts a Regular Expression to NFA, then to DFA, and shows step-by-step transitions.
- **Syntax Tree Visualizer :** Build a tool that takes a grammar and input string and constructs a parse tree or syntax tree dynamically.
- **Code Optimizer Tool :** Implement a small intermediate code optimizer that applies constant folding, dead code elimination, and strength reduction.
- **Grammar Ambiguity Checker :** A tool that checks whether a given CFG is ambiguous and highlights multiple parse paths if any.

Appendix: C Programming Essentials for Compiler Design

This appendix provides a quick reference for basic C programming concepts, functions, and techniques frequently used throughout the programming exercises in this book. It is intended to help readers revise core ideas, avoid common pitfalls, and write cleaner and more reliable code.

A.1 Commonly Used C Functions

These are some of the standard C library functions used in string and character processing throughout this book:

Function	Header	Purpose
isalpha(c)	<ctype.h>	Checks if c is an alphabet
isdigit(c)	<ctype.h>	Checks if c is a digit
tolower(c)	<ctype.h>	Converts c to lowercase
strlen(s)	<string.h>	Returns length of string s
strcpy(dest, src)	<string.h>	Copies src to dest
strcmp(s1, s2)	<string.h>	Compares two strings
strtok(str, delim)	<string.h>	Splits string using delimiter

A.2 Input Handling in C

Input handling in C can be tricky, especially when dealing with multiple words, newlines, or different data types. Here are reliable methods and formats used throughout the book:

1. `scanf("%s", str)` — Reads One Word

- Reads input until the first whitespace.
- Does **not** accept spaces.

```
char str[100];
scanf("%s", str); //Input:Hello World → stores only "Hello"
```

2. `scanf("%[^\n]", str)` — Reads Full Line Until Newline

- Reads until a newline (\n) is encountered.
- Useful for reading full sentences or lines with spaces.

```
char str[100];
scanf("%[^\n]", str);
//Input: Hello World → stores "Hello World"
```

The [^] pattern in scanf tells it to keep reading any character except those listed inside.

3. `fgets(str, sizeof(str), stdin)` — Safe Full-Line Input

- Reads a whole line including spaces.
- Safer than gets() (which is dangerous and deprecated).
- Keeps the newline at the end, so you may want to strip it.

```
fgets(str, sizeof(str), stdin);
str[strcspn(str, "\n")] = '\0'; // Remove newline if needed
```

4. Mixing scanf and fgets Carefully

After using scanf, the newline character remains in the input buffer. This can interfere with fgets().

```
scanf("%d", &n);
getchar(); // Clear newline
fgets(str, sizeof(str), stdin);
```

5. Read Until Specific Characters

Use custom patterns in scanf:

```
scanf("%[^,]", str); // Read until a comma is encountered
```

A.3 Common C Pitfalls

1. **Uninitialized variables:** Always initialize variables before use to avoid undefined behavior.
2. **Buffer Overflow:** Ensure arrays have enough space, and use safer functions like fgets instead of gets.
3. **Missing newline handling:** After using fgets, manually strip newline if needed : `str[strcspn(str, "\n")] = '\0';`
4. **Off-by-One Errors:** Always verify loop bounds carefully to prevent accessing out-of-bound elements.

A.4 File System and Compilation Workflow in C

Understanding how a C program is transformed from human-readable code into an executable file is essential for mastering both compilation and debugging.

1. Source File (.c)

This is the file where you write your C code.

```
main.c
```

It contains your logic using functions like main(), printf(), etc.

2. Compilation → Object File (.o or .obj)

When you compile the C file, the compiler first translates it into an object file, which contains machine code but is not yet executable.

```
gcc -c main.c -o main.o
```

This step checks for syntax errors. The **.o** file holds compiled code, but without linking external libraries or functions.

3. Linking → Executable File (.exe)

The object file is then linked with other necessary code (e.g., from libraries or multiple .o files), producing the final executable.

```
gcc main.o -o main.exe
```

One-Step Compilation (Compile + Link Together)

In most use cases, both steps are done together:

```
gcc main.c -o main.exe
```

Behind the Scenes

A C compilation typically involves four phases:

1. **Preprocessing** → Expand macros and #includes.
2. **Compilation** → Convert to assembly.
3. **Assembly** → Generate machine code (object file).
4. **Linking** → Combine object files into a complete program.

Terminated

Compiler Forge

Unlock the world of compilers with this all-in-one guide to Compiler Design theory and lab! Designed for students and self-learners, this book blends core compiler concepts with real-world programming exercises, complete with GitHub-linked code and hands-on tools. Whether you're battling with grammar rules or debugging lexical analyzers, this book simplifies the journey.

- Covers both theory & lab
- Includes exercises and programs
- GitHub repository for all coding problems
- Integrated Compiler Forge Toolkit for grammar solving
- C language appendix and compiler file system breakdown
- Suitable for CS students, educators, and compiler enthusiasts

Checkout the GitHub Repository

GitHub

https://github.com/garodiaa/Compiler-Forge-Hanbook-Codes

ABOUT THE AUTHOR

Written by a passionate CS student (**GARODIA**) who built Compiler Forge Toolkit from scratch while learning the theory — combining knowledge, creativity, and code.

SOURAV GARODIA

Compiler Forge

A Compiler Design Toolkit